INSTITUTO TECNOLOGICO DE IZTAPALAPA

MATERIA: Lenguajes y Autómatas II

PROFESOR: Abiel Tomás Parra Hernández

ALUMNO:

*QUINTERO BOLIO ERIK EDUARDO 171080151

*COFRADÍA RODRIGUEZ RODRIGO B. 161080399

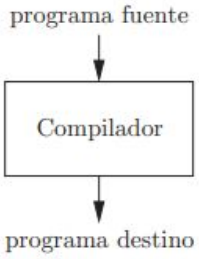
CARRERA: SISTEMAS COMPUTACIONALES

ACTIVIDAD SEMANA 4

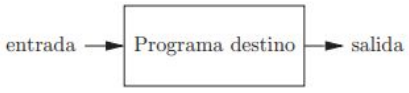
COFRADÍA RODRÍGUEZ RODRIGO BENJAMÍN 161080399 CAPITULO 1 (COMPILADORES PRINCIPIOS, TÉCNICAS Y HERRAMIENTAS)

1.1 Procesadores de lenguaje

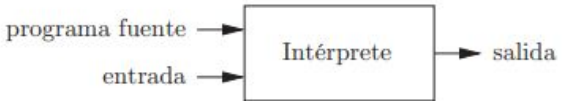
Dicho en forma simple, un compilador es un programa que puede leer un programa en un lenguaje (el lenguaje fuente) y traducirlo en un programa equivalente en otro lenguaje (el lenguaje destino). Una función importante del compilador es reportar cualquier error en el programa fuente que detecte durante el proceso de traducción.



Si el programa destino es un programa ejecutable en lenguaje 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 (fuente) con las entradas proporcionadas por el usuario



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.

1.2 La estructura de un compilador

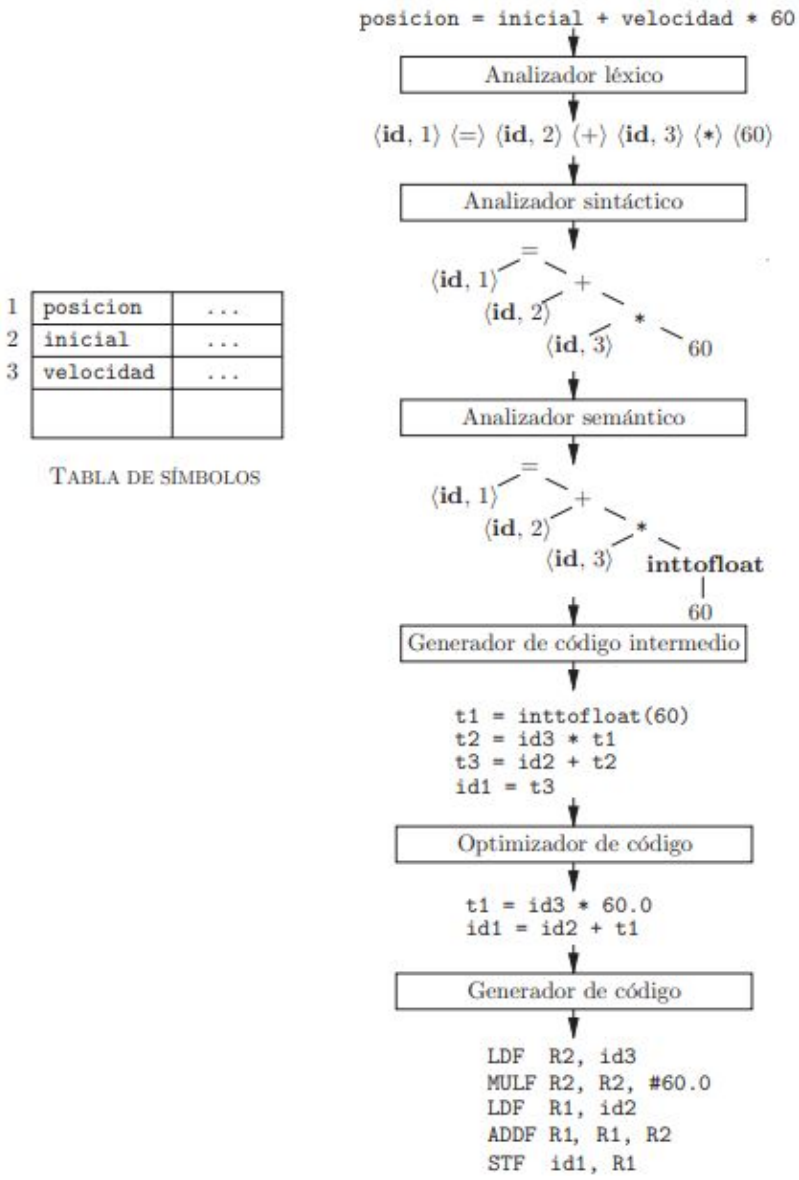
Hasta este punto, hemos tratado al compilador como una caja simple que mapea un programa fuente a un programa destino con equivalencia semántica. Si abrimos esta caja un poco, podremos ver que hay dos procesos en esta asignación: análisis y síntesis.
La parte de la síntesis construye el programa destino 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

1.2.1 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>

1. posicion 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 posicion. La entrada en la tabla de símbolos para un identificador contiene información acerca de éste, como su nombre y tipo.
2. 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.
3. 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.
4. + es un lexema que se asigna al token +.
5. 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.
6. * es un lexema que se asigna al token *.
7. 60 es un lexema que se asigna al token 60. 1

El analizador léxico ignora los espacios en blanco que separan a los lexemas. La figura 1.7 muestra la representación de la instrucción de asignación (1.1) después del análisis léxico como la secuencia de tokens.
< id, 1>< =>< id, 2>< +>< id, 3>< *>< 60>



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

TABLA DE SÍMBOLOS

1.2.2 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.

1.2.3 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.

1.2.4 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

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.

1.2.5 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 (1.3), 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 (intto float) 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

1.2.6 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, 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
```

1.2.7 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.

1.2.8 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. 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 destino.

1.2.9 Herramientas de construcción de compiladores

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.

1.3 La evolución de los lenguajes de programación

Las primeras computadoras electrónicas aparecieron en la década de 1940 y se programaban en lenguaje máquina, mediante secuencias de 0’s y 1’s que indicaban de manera explícita a la computadora las operaciones que debía ejecutar, y en qué orden. Las operaciones en sí eran de muy bajo nivel: mover datos de una ubicación a otra, sumar el contenido de dos registros, comparar dos valores, etcétera. Está demás decir, que este tipo de programación era lenta, tediosa y propensa a errores. Y una vez escritos, los programas eran difíciles de comprender y modificar.

1.3.1 El avance a los lenguajes de alto nivel

El primer paso hacia los lenguajes de programación más amigables para las personas fue el desarrollo de los lenguajes ensambladores a inicios de la década de 1950, los cuales usaban mnemónicos. Al principio, las instrucciones en un lenguaje ensamblador eran sólo representaciones mnemónicas de las instrucciones de máquina. Más adelante, se agregaron macro instrucciones a los lenguajes ensambladores, para que un programador pudiera definir abreviaciones parametrizadas para las secuencias de uso frecuente de las instrucciones de máquina.

En las siguientes décadas se crearon muchos lenguajes más con características innovadoras para facilitar que la programación fuera más natural y más robusta. Más adelante, en este capítulo, hablaremos sobre ciertas características clave que son comunes para muchos lenguajes de programación modernos. En la actualidad existen miles de lenguajes de programación. Pueden clasificarse en una variedad de formas.

1.3.2 Impactos en el compilador

Desde su diseño, los lenguajes de programación y los compiladores están íntimamente relacionados; los avances en los lenguajes de programación impusieron nuevas demandas sobre los escritores de compiladores. Éstos tenían que idear algoritmos y representaciones para traducir y dar soporte a las nuevas características del lenguaje. Desde la década de 1940, la arquitectura de computadoras ha evolucionado también. Los escritores de compiladores no sólo tuvieron que rastrear las nuevas características de un lenguaje, sino que también tuvieron que idear algoritmos de traducción para aprovechar al máximo las nuevas características del hardware. Los compiladores pueden ayudar a promover el uso de lenguajes de alto nivel, al minimizar la sobrecarga de ejecución de los programas escritos en estos lenguajes. Los compiladores también son imprescindibles a la hora de hacer efectivas las arquitecturas computacionales de alto rendimiento en las aplicaciones de usuario. De hecho, el rendimiento de un sistema computacional es tan dependiente de la tecnología de compiladores, que éstos se utilizan como una herramienta para evaluar los conceptos sobre la arquitectura antes de crear una computadora.

Un compilador debe traducir en forma correcta el conjunto potencialmente infinito de programas que podrían escribirse en el lenguaje fuente. El problema de generar el código destino óptimo a partir de un programa fuente es indecidible; por ende, los escritores de compiladores deben evaluar las concesiones acerca de los problemas que se deben atacar y la heurística que se debe utilizar para lidiar con el problema de generar código eficiente.

1.4 La ciencia de construir un compilador

El diseño de compiladores está lleno de bellos ejemplos, en donde se resuelven problemas complicados del mundo real mediante la abstracción de la esencia del problema en forma matemática. Éstos sirven como excelentes ilustraciones de cómo pueden usarse las abstracciones para resolver problemas: se toma un problema, se formula una abstracción matemática que capture las características clave y se resuelve utilizando técnicas matemáticas.

Un compilador debe aceptar todos los programas fuente conforme a la especificación del lenguaje; el conjunto de programas fuente es infinito y cualquier programa puede ser muy largo, posiblemente formado por millones de líneas de código. Cualquier transformación que realice el compilador mientras traduce un programa fuente debe preservar el significado del programa que se está compilando. Por ende, los escritores de compiladores tienen influencia no sólo sobre los compiladores que crean, sino en todos los programas que compilan sus compiladores.

1.4.1 Modelado en el diseño e implementación de compiladores

El estudio de los compiladores es principalmente un estudio de la forma en que diseñamos los modelos matemáticos apropiados y elegimos los algoritmos correctos, al tiempo que logramos equilibrar la necesidad de una generalidad y poder con la simpleza y la eficiencia. Algunos de los modelos más básicos son las máquinas de estados finitos y las expresiones regulares.
Estos modelos son útiles para describir las unidades de léxico de los programas (palabras clave, identificadores y demás) y para describir los algoritmos que utiliza el compilador para reconocer esas unidades. Además, entre los modelos esenciales se encuentran las gramáticas libres de contexto, que se utilizan para describir la estructura sintáctica de los lenguajes de programación, como el anidamiento de los paréntesis o las instrucciones de control.

1.4.2 La ciencia de la optimización de código

El término “optimización” en el diseño de compiladores se refiere a los intentos que realiza un compilador por producir código que sea más eficiente que el código obvio. Por lo tanto, “optimización” es un término equivocado, ya que no hay forma en que se pueda garantizar que el código producido por un compilador sea tan rápido o más rápido que cualquier otro código que realice la misma tarea.

Es difícil, si no es que imposible, construir un compilador robusto a partir de “arreglos”. Por ende, se ha generado una teoría extensa y útil sobre el problema de optimizar código. El uso de una base matemática rigurosa nos permite mostrar que una optimización es correcta y que produce el efecto deseable para todas las posibles entradas.

Las optimizaciones de compiladores deben cumplir con los siguientes objetivos de diseño:
• La optimización debe ser correcta; es decir, debe preservar el significado del programa compilado.
• La optimización debe mejorar el rendimiento de muchos programas.
• El tiempo de compilación debe mantenerse en un valor razonable.
• El esfuerzo de ingeniería requerido debe ser administrable.

1.5 Aplicaciones de la tecnología de compiladores

El diseño de compiladores no es sólo acerca de los compiladores; muchas personas utilizan la tecnología que aprenden al estudiar compiladores en la escuela y nunca, hablando en sentido estricto, han escrito (ni siquiera parte de) un compilador para un lenguaje de programación importante. La tecnología de compiladores tiene también otros usos importantes. Además, el diseño de compiladores impacta en otras áreas de las ciencias computacionales. En esta sección veremos un repaso acerca de las interacciones y aplicaciones más importantes de esta tecnología.

1.5.1 Implementación de lenguajes de programación de alto nivel

Un lenguaje de programación de alto nivel define una abstracción de programación: el programador expresa un algoritmo usando el lenguaje, y el compilador debe traducir el programa en el lenguaje de destino. Por lo general, es más fácil programar en los lenguajes de programación de alto nivel, aunque son menos eficientes; es decir, los programas destino se ejecutan con más lentitud. Los programadores que utilizan un lenguaje de bajo nivel tienen más control sobre un cálculo y pueden, en principio, producir código más eficiente. Por desgracia, los programas de menor nivel son más difíciles de escribir y (peor aún) menos portables, más propensos a errores y más difíciles de mantener.

1.5.2 Optimizaciones para las arquitecturas de computadoras

La rápida evolución de las arquitecturas de computadoras también nos ha llevado a una insaciable demanda de nueva tecnología de compiladores. Casi todos los sistemas de alto rendimiento aprovechan las dos mismas técnicas básicas: paralelismo y jerarquías de memoria. Podemos encontrar el paralelismo en varios niveles: a nivel de instrucción, en donde varias operaciones se ejecutan al mismo tiempo y a nivel de procesador, en donde distintos subprocesos de la misma aplicación se ejecutan en distintos hilos.

Paralelismo

Todos los microprocesadores modernos explotan el paralelismo a nivel de instrucción. Sin embargo, este paralelismo puede ocultarse al programador. Los programas se escriben como si todas las instrucciones se ejecutaran en secuencia; el hardware verifica en forma

dinámica las dependencias en el flujo secuencial de instrucciones y las ejecuta en paralelo siempre que sea posible. En algunos casos, la máquina incluye un programador (scheduler) de hardware que puede modificar el orden de las instrucciones para aumentar el paralelismo en el programa. Ya sea que el hardware reordene o no las instrucciones, los compiladores pueden reordenar las instrucciones para que el paralelismo a nivel de instrucción sea más efectivo.

Jerarquías de memoria

Las jerarquías de memoria se encuentran en todas las máquinas. Por lo general, un procesador tiene un pequeño número de registros que consisten en cientos de bytes, varios niveles de caché que contienen desde kilobytes hasta megabytes, memoria física que contiene desde megabytes hasta gigabytes y, por último, almacenamiento secundario que contiene gigabytes y mucho más. De manera correspondiente, la velocidad de los accesos entre los niveles adyacentes de la jerarquía puede diferir por dos o tres órdenes de magnitud.

1.5.3 Diseño de nuevas arquitecturas de computadoras

En los primeros días del diseño de arquitecturas de computadoras, los compiladores se desarrollaron después de haber creado las máquinas. Eso ha cambiado. Desde que la programación en lenguajes de alto nivel es la norma, el rendimiento de un sistema computacional se determina no sólo por su velocidad en general, sino también por la forma en que los compiladores pueden explotar sus características.

RISC

Uno de los mejores ejemplos conocidos sobre cómo los compiladores influenciaron el diseño de la arquitectura de computadoras fue la invención de la arquitectura RISC (Reduced Instruction-Set Computer, Computadora con conjunto reducido de instrucciones). Antes de esta invención, la tendencia era desarrollar conjuntos de instrucciones cada vez más complejos, destinados a facilitar la programación en ensamblador; estas arquitecturas se denominaron CISC (Complex Instruction-Set Computer, Computadora con conjunto complejo de instrucciones). Por ejemplo, los conjuntos de instrucciones CISC incluyen modos de direccionamiento de memoria complejos para soportar los accesos a las estructuras de datos, e instrucciones para invocar procedimientos que guardan registros y pasan parámetros en la pila

Arquitecturas especializadas

El desarrollo de cada uno de estos conceptos arquitectónicos se acompañó por la investigación y el desarrollo de la tecnología de compiladores correspondiente. Algunas de estas ideas han incursionado en los diseños de las máquinas enbebidas. Debido a que pueden caber sistemas completos en un solo chip, los procesadores ya no necesitan ser unidades primarias preempaquetadas, sino que pueden personalizarse para lograr una

mejor efectividad en costo para una aplicación específica. Por ende, a diferencia de los procesadores de propósito general, en donde las economías de escala han llevado a las arquitecturas computacionales a convergir, los procesadores de aplicaciones específicas exhiben una diversidad de arquitecturas computacionales.

1.5.4 Traducciones de programas

Traducción binaria La tecnología de compiladores puede utilizarse para traducir el código binario para una máquina al código binario para otra máquina distinta, con lo cual se permite a una máquina ejecutar los programas que originalmente eran compilados para otro conjunto de instrucciones. Varias compañías de computación han utilizado la tecnología de la traducción binaria para incrementar la disponibilidad de software en sus máquinas.

Síntesis de hardware

No sólo la mayoría del software está escrito en lenguajes de alto nivel; incluso hasta la mayoría de los diseños de hardware se describen en lenguajes de descripción de hardware de alto nivel, como Verilog y VHDL (Very High-Speed Integrated Circuit Hardware Description Lenguaje, Lenguaje de descripción de hardware de circuitos integrados de muy alta velocidad). Por lo general, los diseños de hardware se describen en el nivel de transferencia de registros (RTL), en donde las variables representan registros y las expresiones representan la lógica combinacional. Las herramientas de síntesis de hardware traducen las descripciones RTL de manera automática en compuertas, las cuales a su vez se asignan a transistores y, en un momento dado, a un esquema físico

Simulación compilada

La simulación es una técnica general que se utiliza en muchas disciplinas científicas y de ingeniería para comprender un fenómeno, o para validar un diseño. Por lo general, las entradas de los simuladores incluyen la descripción del diseño y los parámetros específicos de entrada para esa ejecución específica de la simulación. Las simulaciones pueden ser muy costosas. Por lo general, necesitamos simular muchas alternativas de diseño posibles en muchos conjuntos distintos de entrada, y cada experimento puede tardar días en completarse, en una máquina de alto rendimiento. En vez de escribir un simulador para interpretar el diseño, es más rápido compilar el diseño para producir código máquina que simule ese diseño específico en forma nativa. La simulación compilada puede ejecutarse muchos grados de magnitud más rápido que un método basado en un intérprete.

1.5.5 Herramientas de productividad de software

Sin duda, los programas son los artefactos de ingeniería más complicados que se hayan producido jamás; consisten en muchos, muchos detalles, cada uno de los cuales debe corregirse para que el programa funcione por completo. Como resultado, los errores proliferan en los programas; éstos pueden hacer que un sistema falle, producir resultados incorrectos, dejar un sistema vulnerable a los ataques de seguridad, o incluso pueden llevar a fallas catastróficas en sistemas críticos. La prueba es la técnica principal para localizar errores en los programas. Un enfoque complementario interesante y prometedor es utilizar el análisis de flujos de datos para localizar errores de manera estática (es decir, antes de que se ejecute el programa). El análisis de flujos de datos puede buscar errores a lo largo de todas las rutas posibles de ejecución, y no sólo aquellas ejercidas por los conjuntos de datos de entrada, como en el caso del proceso de prueba de un programa.

Comprobación (verificación) de tipos

La comprobación de tipos es una técnica efectiva y bien establecida para captar las inconsistencias en los programas. Por ejemplo, puede usarse para detectar errores en donde se aplique una operación al tipo incorrecto de objeto, o si los parámetros que se pasan a un procedimiento no coinciden con su firma. El análisis de los programas puede ir más allá de sólo encontrar los errores de tipo, analizando el flujo de datos a través de un programa. Por ejemplo, si a un apuntador se le asigna null y se desreferencia justo después, es evidente que el programa tiene un error. La misma tecnología puede utilizarse para detectar una variedad de huecos de seguridad, en donde un atacante proporciona una cadena u otro tipo de datos que el programa utiliza sin cuidado.

Comprobación de límites

Es más fácil cometer errores cuando se programa en un lenguaje de bajo nivel que en uno de alto nivel. Por ejemplo, muchas brechas de seguridad en los sistemas se producen debido a los desbordamientos en las entradas y salidas de de los programas escritos en C. Como C no comprueba los límites de los arreglos, es responsabilidad del usuario asegurar que no se acceda a los arreglos fuera de los límites. Si no se comprueba que los datos suministrados por el usuario pueden llegar a desbordar un elemento, el programa podría caer en el truco de almacenar los datos del usuario fuera del espacio asociado a este elemento. Un atacante podría manipular los datos de entrada que hagan que el programa se comporte en forma errónea y comprometa la seguridad del sistema. Se han desarrollado técnicas para encontrar los desbordamientos de búfer en los programas, pero con un éxito limitado

1.6 Fundamentos de los lenguajes de programación

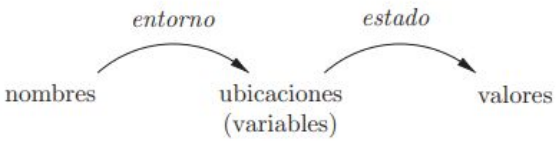
En esta sección hablaremos sobre la terminología más importante y las distinciones que aparecen en el estudio de los lenguajes de programación. No es nuestro objetivo abarcar todos los conceptos o todos los lenguajes de programación populares. Asumimos que el lector está familiarizado por lo menos con uno de los lenguajes C, C++, C# o Java, y que tal vez conozca otros.

1.6.1 La distinción entre estático y dinámico

Una de las cuestiones más importantes a las que nos enfrentamos al diseñar un compilador para un lenguaje es la de qué decisiones puede realizar el compilador acerca de un programa. Si un lenguaje utiliza una directiva que permite al compilador decidir sobre una cuestión, entonces decimos que el lenguaje utiliza una directiva estática, o que la cuestión puede decidirse en tiempo de compilación. Por otro lado, se dice que una directiva que sólo permite realizar una decisión a la hora de ejecutar el programa es una directiva dinámica, o que requiere una decisión en tiempo de ejecución

1.6.2 Entornos y estados

Otra distinción importante que debemos hacer al hablar sobre los lenguajes de programación es si los cambios que ocurren a medida que el programa se ejecuta afectan a los valores de los elementos de datos, o si afectan a la interpretación de los nombres para esos datos. Por ejemplo, la ejecución de una asignación como x = y + 1 cambia el valor denotado por el nombre x. Dicho en forma más específica, la asignación cambia el valor en cualquier ubicación denotada por x. Tal vez sea menos claro que la ubicación denotada por x puede cambiar en tiempo de ejecución. Por ejemplo, como vimos en el ejemplo 1.3, si x no es una variable estática (o de “clase”), entonces cada objeto de la clase tiene su propia ubicación para una instancia de la variable x.



1.6.3 Alcance estático y estructura de bloques

La mayoría de los lenguajes, incluyendo a C y su familia, utilizan el alcance estático. Las reglas de alcance para C se basan en la estructura del programa; el alcance de una declaración se determina en forma implícita, mediante el lugar en el que aparece la declaración en el programa. Los lenguajes posteriores, como C++, Java y C#, también proporcionan un control explícito sobre los alcances, a través del uso de palabras clave como public, private y protected. En esta sección consideramos las reglas de alcance

estático para un lenguaje con bloques, en donde un bloque es una agrupación de declaraciones e instrucciones. C utiliza las llaves { y } para delimitar un bloque; el uso alternativo de begin y end para el mismo fin se remonta hasta Algol.

1.6.4 Control de acceso explícito

Las clases y las estructuras introducen un nuevo alcance para sus miembros. Si p es un objeto de una clase con un campo (miembro) x, entonces el uso de x en p.x se refiere al campo x en la definición de la clase. En analogía con la estructura de bloques, el alcance de la declaración de un miembro x en una clase C se extiende a cualquier subclase C , excepto si C tiene una declaración local del mismo nombre x. Mediante el uso de palabras clave como public, private y protected, los lenguajes orientados a objetos como C++ o Java proporcionan un control explícito sobre el acceso a los nombres de los miembros en una superclase. Estas palabras clave soportan el encapsulamiento mediante la restricción del acceso. Por ende, los nombres privados reciben de manera intencional un alcance que incluye sólo las declaraciones de los métodos y las definiciones asociadas con esa clase, y con cualquier clase “amiga” (friend: el término de C++).

1.6.5 Alcance dinámico

Técnicamente, cualquier directiva de alcance es dinámica si se basa en un factor o factores que puedan conocerse sólo cuando se ejecute el programa. Sin embargo, el término alcance dinámico se refiere, por lo general, a la siguiente directiva: el uso de un nombre x se refiere a la declaración de x en el procedimiento que se haya llamado más recientemente con dicha declaración. El alcance dinámico de este tipo aparece sólo en situaciones especiales. Vamos a considerar dos ejemplos de directivas dinámicas: la expansión de macros en el preprocesador de C y la resolución de métodos en la programación orientada a objetos.

1.6.6 Mecanismos para el paso de parámetros

Todos los lenguajes de programación tienen una noción de un procedimiento, pero pueden diferir en cuanto a la forma en que estos procedimientos reciben sus argumentos. En esta sección vamos a considerar cómo se asocian los parámetros actuales (los parámetros que se utilizan en la llamada a un procedimiento) con los parámetros formales (los que se utilizan en la definición del procedimiento).

Llamada por valor

En la llamada por valor, el parámetro actual se evalúa (si es una expresión) o se copia (si es una variable). El valor se coloca en la ubicación que pertenece al correspondiente parámetro formal del procedimiento al que se llamó. Este método se utiliza en C y en Java, además de ser una opción común en C++, así como en la mayoría de los demás lenguajes. La llamada por valor tiene el efecto de que todo el cálculo que involucra a los parámetros

formales, y que realiza el procedimiento al que se llamó, es local para ese procedimiento, y los parámetros actuales en sí no pueden modificarse

Llamada por nombre

Hay un tercer mecanismo (la llamada por nombre) que se utilizó en uno de los primeros lenguajes de programación: Algol 60. Este mecanismo requiere que el procedimiento al que se llamó se ejecute como si el parámetro actual se sustituyera literalmente por el parámetro formal en su código, como si el procedimiento formal fuera una macro que representa al parámetro actual (cambiando los nombres locales en el procedimiento al que se llamó, para que sean distintos).

1.6.7 Uso de alias

Hay una consecuencia interesante del paso por parámetros tipo llamada por referencia o de su simulación, como en Java, en donde las referencias a los objetos se pasan por valor. Es posible que dos parámetros formales puedan referirse a la misma ubicación; se dice que dichas variables son alias una de la otra. Como resultado, dos variables cualesquiera, que dan la impresión de recibir sus valores de dos parámetros formales distintos, pueden convertirse en alias una de la otra, también.

APUNTES VIDEO 1 (Mod-01 Lec-01 An Overview of a Compiler)

APLICACIONES DE LA TECNOLOGÍA DE COMPILADORES

* IMPLEMENTACIÓN DE LENGUAJES DE ALTO NIVEL.

*TRADUCCIONES DE PROGRAMAS.

*GENERACIÓN DE CÓDIGO MÁQUINA PARA LENGUAJES DE ALTO NIVEL.

*HERRAMIENTAS DE PRODUCTIVIDAD DE SOFTWARE.

*PRUEBA DE SOFTWARE.

*DISEÑO DE ARQUITECTURA DE COMPUTADORAS.

*DISEÑO DE DETECCIÓN DE CÓDIGO MALICIOSO PARA NUEVAS ARQUITECTURAS DE COMPUTADORAS.

*INTÉRPRETES PARA JAVASCRIPT Y FLASH.

COMPLEJIDAD DE LA TECNOLOGÍA DE COMPILADORES

*Utiliza algoritmos y técnicas de un gran número de áreas de informática.

*Traduce la teoría compleja a la práctica.

* Es el software de sistema más complejo.

SISTEMA DE PROCESAMIENTO DEL LENGUAJE

programa fuente

1.- PREPROCESADOR

Programa fuente final

2.- COMPILADOR

Código ensamblador

3.- ENSAMBLADOR

Objetos en código máquina

4.- ENLAZADOR

Programa final

ETAPAS DE UN COMPILADOR

Programa fuente

1.- ANALIZADOR LÉXICO

Tokens

2.- ANALIZADOR SINTÁCTICO

Árbol sintáctico

3.- ANALIZADOR SEMÁNTICO

Árbol sintáctico

4.- GENERADOR DE CÓDIGO OBJETO

Código intermedio

5.- OPTIMIZADOR DE CÓDIGO

Código intermedio

6.- GENERADOR DE CÓDIGO OBJETO

Código objeto

APUNTES DEL SEGUNDO VIDEO (*Mod-02 Lec-02 Lexical Analysis)*

El análisis léxico es la primera fase de un compilador, tiene como entrada el código fuente en cualquier lenguaje de programación el cual es leído carácter por carácter por el compilador y éste mismo nos da como salida componentes léxicos o tokens, que son posteriormente proporcionados al analizador sintáctico.

También nos habla sobre la importancia de resaltar las diferencia entre el análisis léxico y el análisis sintáctico, dándonos como principal razón una simplificación del diseño y mejora de eficiencia de un compilador que va de la mano de la optimización y gestión de un software usando la ingeniería de software.

Los tokens, patrones y lexemas, los componente léxicos o tokens se definen como una secuencia de caracteres con significado sintáctico propio y que son pertenecientes a una categoría léxica (identificador, palabra reservada, literales, operadores o caracteres de puntuación) y estos pueden contener uno o mas lexemas. El lexema es una secuencia de caracteres cuya estructura se corresponde con el patrón de un token y los patrones son la regla que describe los lexemas correspondientes a un token. El patrón es la regla que describe el conjunto de lexemas que pueden representar a un determinado componente léxico en los programas fuente. En otras palabras, es la descripción del componente léxico mediante una regla.

El análisis léxico no es perfecto y tiene sus restricciones o dificultades, para ejemplificar y demostrarlos no pone el ejemplo de las palabras reservadas, ponen de ejemplo a C y PL/1, C tiene las palabras reservadas como while, do, if, else, las cuales PL/1 no contiene, por lo que un compilador diseñado para el lenguaje C no reconocería de manera correcta los tokens en PL/1. El análisis léxico no puede detectar ningún error significativo, excepto errores simples, como símbolos ilegales y otros más simples.

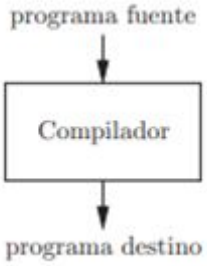
El reconocimiento y especificación de tokens nos habla que se puede realizar mediante un autómata finito, un autómata finito (FA) es una máquina abstracta simple que se utiliza para reconocer patrones dentro de la entrada tomada de algún conjunto de caracteres (o alfabeto) tomando como ejemplo el lenguaje C. El trabajo de un FA es aceptar o rechazar una entrada dependiendo de si el patrón definido por FA ocurre en la entrada.

Quintero Bolio Erik Eduardo - 171080151 - ISC-7AM - Resumen capitulo 1- individual

Los lenguajes de programación son notaciones que describen los cálculos a las personas y las máquinas. Los programas encargados de traducciones de lenguaje humano a máquina se llaman compiladores.

Procesadores de lenguaje

un compilador es un programa que puede leer un programa en un lenguaje (el lenguaje fuente) y traducirlo en un programa equivalente en otro lenguaje (el lenguaje destino)

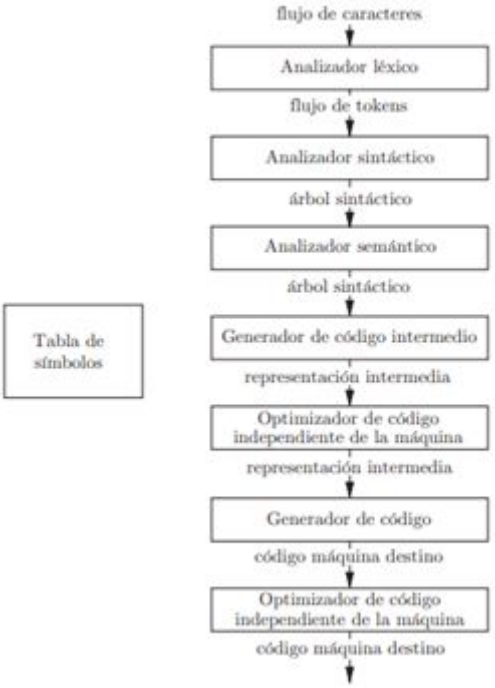


Un intérprete es otro tipo común de procesador de lenguaje. En vez de producir un programa destino como una traducción, el intérprete nos da la apariencia de ejecutar directamente las operaciones especificadas en el programa de origen (fuente) con las entradas proporcionadas por el usuario

La estructura de un compilador

La parte del análisis divide el programa fuente en componentes e impone una estructura gramatical sobre ellas 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, esta es comúnmente llamada front-end.

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, esta es comúnmente llamada back-end.



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

Análisis sintáctico

La segunda fase del compilador es el análisis sintáctico o parsing. El parser 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.

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.

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

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. 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.

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.

Administración de la tabla de símbolos

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.

El agrupamiento de fases en pasadas

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.

Herramientas de construcción de compiladores

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.

La evolución de los lenguajes de programación

Las primeras computadoras electrónicas aparecieron en la década de 1940 y se programaban en lenguaje máquina, mediante secuencias de 0’s y 1’s que indicaban de manera explícita a la computadora las operaciones que debía ejecutar, y en qué orden.

El avance a los lenguajes de alto nivel

El primer paso hacia los lenguajes de programación más amigables para las personas fue el desarrollo de los lenguajes ensambladores a inicios de la década de 1950, los cuales usaban mnemónicos. Un paso importante hacia los lenguajes de alto nivel se hizo en la segunda mitad de la década de 1950, con el desarrollo de Fortran para la computación científica. En las siguientes décadas se crearon muchos lenguajes más con características innovadoras para facilitar que la programación fuera más natural y más robusta.

El término lenguaje von Neumann se aplica a los lenguajes de programación cuyo modelo se basa en la arquitectura de computadoras descrita por von Neumann.

Un lenguaje orientado a objetos es uno que soporta la programación orientada a objetos, un estilo de programación en el que un programa consiste en una colección de objetos que interactúan entre sí.

Impactos en el compilador

Los compiladores pueden ayudar a promover el uso de lenguajes de alto nivel, al minimizar la sobrecarga de ejecución de los programas escritos en estos lenguajes. Los compiladores también son imprescindibles a la hora de hacer efectivas las arquitecturas computacionales de alto rendimiento en las aplicaciones de usuario.

Un compilador debe traducir en forma correcta el conjunto potencialmente infinito de programas que podrían escribirse en el lenguaje fuente.

La ciencia de construir un compilador

Un compilador debe aceptar todos los programas fuente conforme a la especificación del lenguaje; el conjunto de programas fuente es infinito y cualquier programa puede ser muy largo, posiblemente formado por millones de líneas de código.

Modelado en el diseño e implementación de compiladores

Algunos de los modelos más básicos son las máquinas de estados finitos y las expresiones regulares, estos modelos son útiles para describir las unidades de léxico de los programas y para describir los algoritmos que utiliza el compilador para reconocer esas unidades.

La ciencia de la optimización de código

Las optimizaciones de compiladores deben cumplir con los siguientes objetivos de diseño:

- La optimización debe ser correcta; es decir, debe preservar el significado del programa compilado.
- La optimización debe mejorar el rendimiento de muchos programas.
- El tiempo de compilación debe mantenerse en un valor razonable.
- El esfuerzo de ingeniería requerido debe ser administrable.

al estudiar los compiladores no sólo aprendemos a construir uno, sino también la metodología general para resolver problemas complejos y abiertos.

Aplicaciones de la tecnología de compiladores

La tecnología de compiladores tiene también otros usos importantes.

Implementación de lenguajes de programación de alto nivel

Un lenguaje de programación de alto nivel define una abstracción de programación: el programador expresa un algoritmo usando el lenguaje, y el compilador debe traducir el programa

en el lenguaje de destino. La orientación a objetos se introdujo por primera vez en Simula en 1967, y se ha incorporado en lenguajes como Smalltalk, C++, C# y Java.

Optimizaciones para las arquitecturas de computadoras

Casi todos los sistemas de alto rendimiento aprovechan las dos mismas técnicas básicas: paralelismo y jerarquías de memoria. Podemos encontrar el paralelismo en varios niveles: a nivel de instrucción, en donde varias operaciones se ejecutan al mismo tiempo y a nivel de procesador, en donde distintos subprocesos de la misma aplicación se ejecutan en distintos hilos.

En el paralelismo todos los programas se escriben como si todas las instrucciones se ejecutaran en secuencia; el hardware verifica en forma dinámica las dependencias en el flujo secuencial de instrucciones y las ejecuta en paralelo siempre que sea posible.

Una jerarquía de memoria consiste en varios niveles de almacenamiento con distintas velocidades y tamaños, en donde el nivel más cercano al procesador es el más rápido, pero también el más pequeño. Las jerarquías de memoria se encuentran en todas las máquinas. Por lo general, un procesador tiene un pequeño número de registros que consisten en cientos de bytes

Diseño de nuevas arquitecturas de computadoras

El desarrollo de arquitecturas de computadoras modernas, los compiladores se desarrollan en la etapa de diseño del procesador, y se utiliza el código compilado, que se ejecuta en simuladores, para evaluar las características propuestas sobre la arquitectura.

Uno de los mejores ejemplos conocidos sobre cómo los compiladores influenciaron el diseño de la arquitectura de computadoras fue la invención de la arquitectura RISC, la mayoría de las arquitecturas de procesadores de propósito general, como PowerPC, SPARC, MIPS, Alpha y PA-RISC, se basan en el concepto RISC.

En las últimas tres décadas se han propuesto muchos conceptos sobre la arquitectura, entre los cuales se incluyen las máquinas de flujo de datos, las máquinas VLIW, los arreglos SIMD de procesadores, los arreglos sistólicos, los multiprocesadores con memoria compartida y los multiprocesadores con memoria distribuida.

Traducciones de programas

Existen técnicas de traducción de programas

Traducción binaria puede utilizarse para traducir el código binario para una máquina al código binario para otra máquina distinta, con lo cual se permite a una máquina ejecutar los programas que originalmente eran compilados para otro conjunto de instrucciones.

Síntesis de hardware, sus herramientas de síntesis de hardware traducen las descripciones RTL de manera automática en compuertas, las cuales a su vez se asignan a transistores y, en un momento dado, a un esquema físico.

Interpretes de consultas en base de datos se utilizan para realizar búsquedas en bases de datos. Las consultas en las bases de datos consisten en predicados que contienen operadores relacionales y booleanos.

La simulación es una técnica general que se utiliza en muchas disciplinas científicas y de ingeniería para comprender un fenómeno, o para validar un diseño.

Herramientas de productividad de software

los programas son los artefactos de ingeniería más complicados que se hayan producido jamás; consisten en muchos, muchos detalles, cada uno de los cuales debe corregirse para que el programa funcione por completo. La prueba es la técnica principal para localizar errores en los programas. El problema de encontrar todos los errores en los programas es indicidible. Puede diseñarse un análisis de flujos de datos para advertir a los programadores acerca de todas las posibles instrucciones que violan una categoría específica de errores.

La comprobación de tipos es una técnica efectiva y bien establecida para captar las inconsistencias en los programas.

comprueba los límites de los arreglos, es responsabilidad del usuario asegurar que no se acceda a los arreglos fuera de los límites. Si no se comprueba que los datos suministrados por el usuario pueden llegar a desbordar un elemento, el programa podría caer en el truco de almacenar los datos del usuario fuera del espacio asociado a este elemento.

La administración automática de la memoria elimina todos los errores de administración de memoria

Fundamentos de los lenguajes de programación

Terminología más importante y las distinciones que aparecen en el estudio de los lenguajes de programación.

La distinción entre estático y dinámico

Una de las cuestiones en las que nos debemos de concentrar es en el alcance de las declaraciones. El alcance de una declaración de x es la región del programa en la que los usos de x se refieren a esta declaración. Un lenguaje utiliza el alcance estático o alcance léxico si es posible determinar el alcance de una declaración con sólo ver el programa. En cualquier otro caso, el lenguaje utiliza un alcance dinámico.

Entornos y estados

Otra distinción importante que debemos hacer al hablar sobre los lenguajes de programación es si los cambios que ocurren a medida que el programa se ejecuta afectan a los valores de los elementos de datos, o si afectan a la interpretación de los nombres para esos datos.

Alcance estático y estructura de bloques

consideramos las reglas de alcance estático para un lenguaje con bloques, en donde un bloque es una agrupación de declaraciones e instrucciones. C utiliza las llaves { y } para delimitar un bloque.

Control de acceso explícito

Las clases y las estructuras introducen un nuevo alcance para sus miembros. Si p es un objeto de una clase con un campo (miembro) x, entonces el uso de x en p.x se refiere al campo x en la definición de la clase.

Alcance dinámico

Técnicamente, cualquier directiva de alcance es dinámica si se basa en un factor o factores que puedan conocerse sólo cuando se ejecute el programa, este tipo aparece sólo en situaciones especiales.

Mecanismos para el paso de parámetros

El mecanismo que se utilice será el que determine la forma en que el código de secuencia de llamadas tratará a los parámetros.

En la llamada por valor, el parámetro actual se evalúa (si es una expresión) o se copia (si es una variable). El valor se coloca en la ubicación que pertenece al correspondiente parámetro formal del procedimiento al que se llamó.

En la llamada por referencia, la dirección del parámetro actual se pasa al procedimiento al que se llamó como el valor del correspondiente parámetro formal.

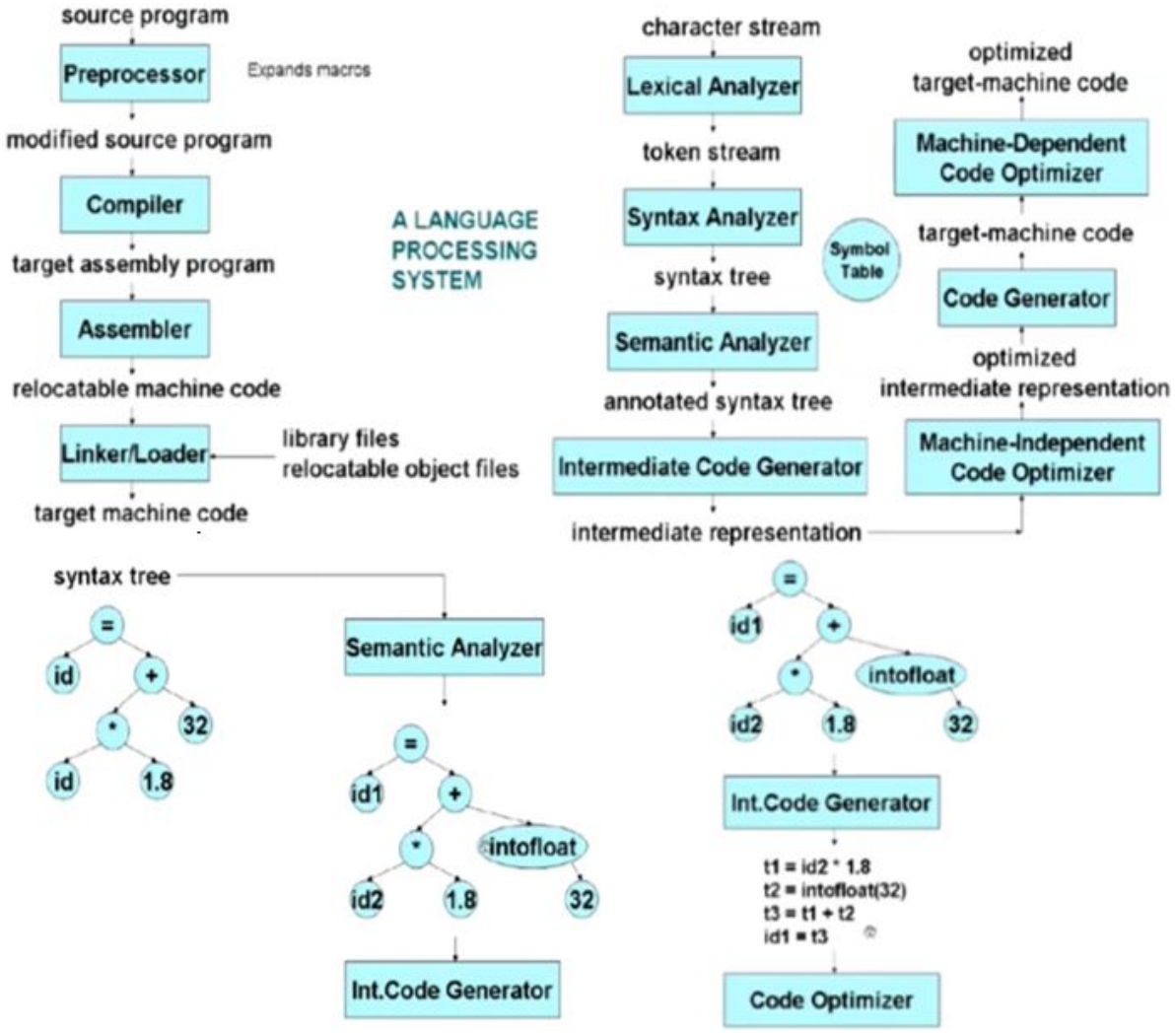
Hay un tercer mecanismo (la llamada por nombre) que se utilizó en uno de los primeros lenguajes de programación.

Uso de alias

Es posible que dos parámetros formales puedan referirse a la misma ubicación; se dice que dichas variables son alias una de la otra.

Mod-01 Lec-01

En este libro se habla básicamente de lo mismo que trato el libro por lo que de manera personal no realizare un resumen de este más que lo dedicado en la siguiente imagen que fue una recopilación de las diapositivas que tenía el video, esto lo hago con el afán de ejemplificar y complementar así lo visto en el libro ya que ahí le hizo falta este tipo de diagramas

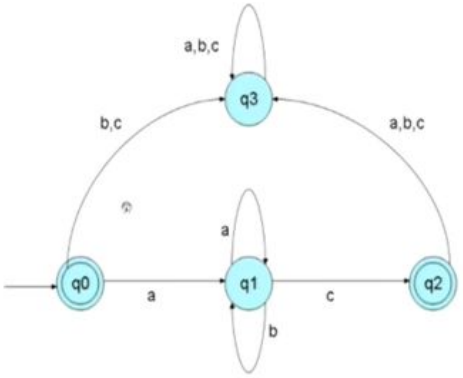


Mod-02 Lec-02

para este caso es lo mismo, siento que el libro lo explica mejor, tal vez sea porque no todo le entiendo claramente puesto que el inglés es algo tecnico y me cuesta trabajo pero en este si siento importante enfatizar lo que se menciona de las expresiones regulares

Las expresiones regulares son patrones utilizados para encontrar una determinada combinación de caracteres dentro de una cadena de texto. Las expresiones regulares proporcionan una manera muy flexible de buscar o reconocer cadenas de texto.

E ilustrarlo con el ejemplo que mejor entendí dentro de este video.



- $Q = \{q_0, q_1, q_2, q_3\}$
- $\Sigma = \{a, b, c\}$
- q_0 is the start state and $F = \{q_0, q_2\}$
- The transition function δ is defined by the table below

state	symbol		
	a	b	c
q_0	q_1	q_3	q_3
q_1	q_1	q_1	q_2
q_2	q_3	q_3	q_3
q_3	q_3	q_3	q_3

The accepted language is the set of all strings beginning with an 'a' and ending with a 'c' (ϵ is also accepted)

TRABAJO EN EQUIPO

LIBRO COMPILADORES

♦ Procesadores de lenguaje. Un entorno de desarrollo integrado de software incluye muchos tipos distintos de procesadores de lenguaje, como compiladores, intérpretes, ensambladores, enlazadores, cargadores, depuradores, profilers.

♦ Fases del compilador. Un compilador opera como una secuencia de fases, cada una de las cuales transforma el programa fuente de una representación intermedia a otra.

♦ Lenguajes máquina y ensamblador. Los lenguajes máquina fueron los lenguajes de programación de la primera generación, seguidos de los lenguajes ensambladores. La programación en estos lenguajes requería de mucho tiempo y estaba propensa a errores.

♦ Modelado en el diseño de compiladores. El diseño de compiladores es una de las fases en las que la teoría ha tenido el mayor impacto sobre la práctica. Entre los modelos que se han encontrado de utilidad se encuentran: autómatas, gramáticas, expresiones regulares, árboles y muchos otros.

♦ Optimización de código. Aunque el código no puede verdaderamente “optimizarse”, esta ciencia de mejorar la eficiencia del código es tanto compleja como muy importante. Constituye una gran parte del estudio de la compilación.

♦ Lenguajes de alto nivel. A medida que transcurre el tiempo, los lenguajes de programación se encargan cada vez más de las tareas que se dejaban antes al programador, como la administración de memoria, la comprobación de consistencia en los tipos, o la ejecución del código en paralelo.

♦ Compiladores y arquitectura de computadoras. La tecnología de compiladores ejerce una influencia sobre la arquitectura de computadoras, así como también se ve influenciada por los avances en la arquitectura. Muchas innovaciones modernas en la arquitectura dependen de la capacidad de los compiladores para extraer de los programas fuente las oportunidades de usar con efectividad las capacidades del hardware.

♦ Productividad y seguridad del software. La misma tecnología que permite a los compiladores optimizar el código puede usarse para una variedad de tareas de análisis de programas, que van desde la detección de errores comunes en los programas, hasta el descubrimiento de que un programa es vulnerable a uno de los muchos tipos de intrusiones que han descubierto los “hackers”.

♦ Reglas de alcance. El alcance de una declaración de x es el contexto en el cual los usos de x se refieren a esta declaración. Un lenguaje utiliza el alcance estático o alcance léxico si es posible determinar el alcance de una declaración con sólo analizar el programa. En cualquier otro caso, el lenguaje utiliza un alcance dinámico.

♦ Entornos. La asociación de nombres con ubicaciones en memoria y después con los valores puede describirse en términos de entornos, los cuales asignan los nombres a las ubicaciones en memoria, y los estados, que asignan las ubicaciones a sus valores.

♦ Estructura de bloques. Se dice que los lenguajes que permiten anidar bloques tienen estructura de bloques. Un nombre x en un bloque anidado B se encuentra en el alcance de una declaración D de x en un bloque circundante, si no existe otra declaración de x en un bloque intermedio.

♦ Paso de parámetros. Los parámetros se pasan de un procedimiento que hace la llamada al procedimiento que es llamado, ya sea por valor o por referencia. Cuando se pasan objetos grandes por valor, los valores que se pasan son en realidad referencias a los mismos objetos, lo cual resulta en una llamada por referencia efectiva.

♦ Uso de alias. Cuando los parámetros se pasan (de manera efectiva) por referencia, dos parámetros formales pueden referirse al mismo objeto. Esta posibilidad permite que un cambio en una variable cambie a la otra.

VIDEOS (Mod-01 Lec-01 An Overview of a Compiler)

SISTEMA DE PROCESAMIENTO DEL LENGUAJE

programa fuente

1.- PREPROCESADOR

Programa fuente final

2.- COMPILADOR

Código ensamblador

3.- ENSAMBLADOR

Objetos en código máquina

4.- ENLAZADOR

Programa final

ETAPAS DE UN COMPILADOR

Programa fuente

1.- ANALIZADOR LÉXICO

Tokens

2.- ANALIZADOR SINTÁCTICO

Árbol sintáctico

3.- ANALIZADOR SEMÁNTICO

Árbol sintáctico

4.- GENERADOR DE CÓDIGO OBJETO

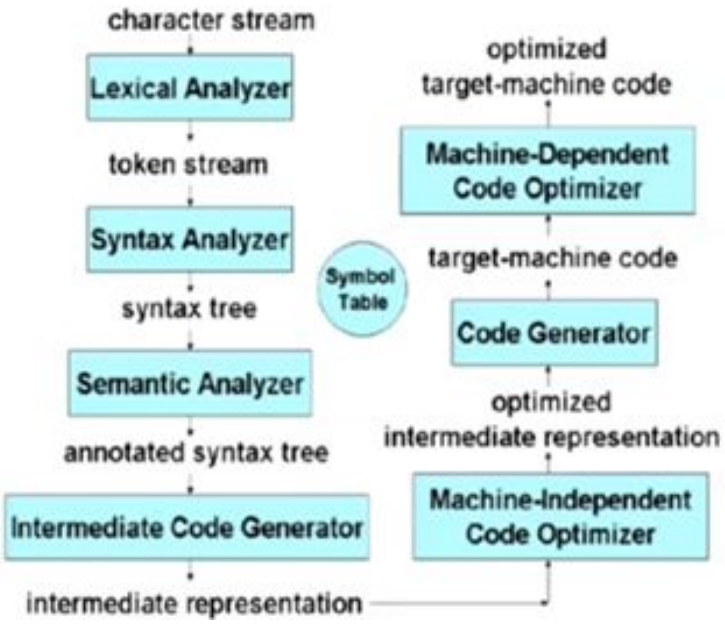
Código intermedio

5.- OPTIMIZADOR DE CÓDIGO

Código intermedio

6.- GENERADOR DE CÓDIGO OBJETO

Código objeto



ANALISIS LEXICO

Es la primera fase de un compilador y consiste en un programa que recibe como entrada el código fuente de otro programa (secuencia de caracteres) y produce una salida compuesta de *tokens* (componentes léxicos) o símbolos. Estos *tokens* sirven para una posterior etapa del proceso de traducción, siendo la entrada para el Análisis Sintáctico.

La especificación de un lenguaje de programación a menudo incluye un conjunto de reglas que definen el léxico. Estas reglas consisten comúnmente en expresiones regulares que indican el conjunto de posibles secuencias de caracteres que definen un *token* o lexema.

En algunos lenguajes de programación es necesario establecer patrones para caracteres especiales (como el espacio en blanco) que la gramática pueda reconocer sin que constituya un *token* en sí.

ANÁLISIS SINTÁCTICO

Es una de las partes de un compilador que transforma su entrada en un árbol de derivación.

El análisis sintáctico convierte el texto de entrada en otras estructuras (comúnmente árboles), que son más útiles para el posterior análisis y capturan la jerarquía implícita de la

entrada. Un analizador crea tokens de una secuencia de caracteres de entrada y son estos tokens los que son procesados por el analizador sintáctico para construir la estructura de datos, por ejemplo un árbol de análisis o árboles de sintaxis abstracta.

ANÁLISIS 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 operando 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.

GENERACION DE CODIGO INTERMEDIO

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.

Al convertir el código fuente en un código intermedio, se puede escribir un optimizador de código independiente de la máquina.

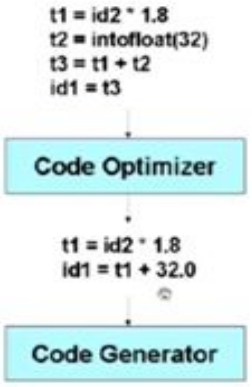
El código intermedio debe ser fácil de producir y fácil de traducir a código de máquina.

OPTIMIZACIÓN DE CÓDIGO INDEPENDIENTE DE LA MÁQUINA

Es el conjunto de fases de un compilador que transforma un fragmento de código en otro fragmento con un comportamiento equivalente y que se ejecuta de forma más eficiente.

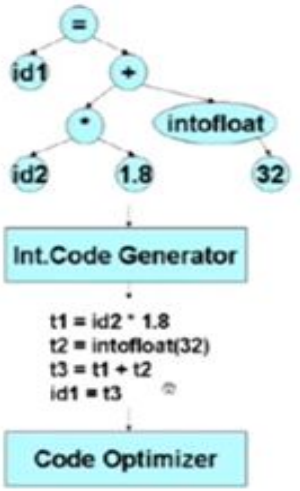
El proceso de generación de código intermedio presenta ineficiencias. La optimización de código elimina tales ineficiencias y mejora el código.

La optimización del código consta de un montón de procesos y el porcentaje de mejora depende de los programas



GENERACION DE CODIGO

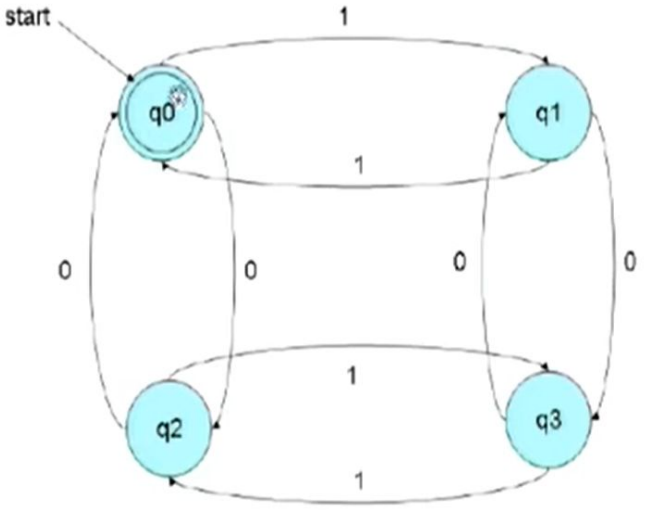
Consiste en código de máquina relocalizable o código ensamblador. Las posiciones de memoria se seleccionan para cada una de las variables usadas por el programa. Después, cada una de las instrucciones intermedias se traduce a una secuencia de instrucciones de máquina que ejecuta la misma tarea.



VIDEO (Mod-02 Lec-02 Lexical Analysis)

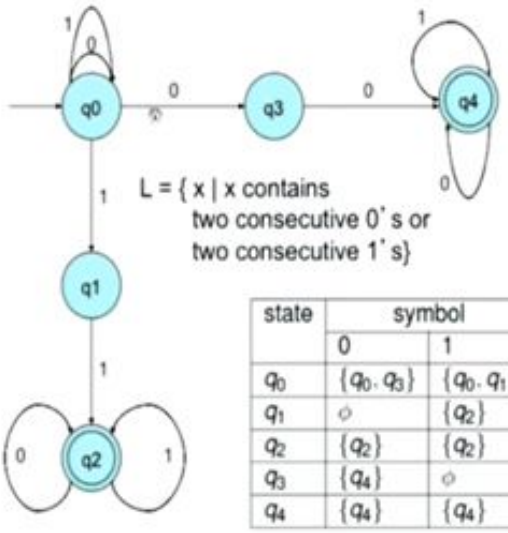
Esta es la primera etapa del compilador. Toma el código fuente de cualquier lenguaje de programación como entrada, que el compilador lee carácter a carácter, y el compilador nos proporciona componentes léxicos de salida o tokens, que luego se proporcionan al analizador.

También habló de la importancia de enfatizar la diferencia entre análisis léxico y análisis sintáctico, lo que nos convierte en una de las principales razones para simplificar el diseño y mejorar la eficiencia del compilador, y el compilador está muy relacionado con la optimización y gestión del software utilizando el software Ingeniería de software.



- $Q = \{q_0, q_1, q_2, q_3\}$, q_0 is the start state
- $F = \{q_0\}$, δ is as in the figure
- Language accepted is the set of all strings of 0's and 1's, in which the no. of 0's and the no. of 1's are even numbers

Nondeterministic FSA Example - 1



state	symbol	
	0	1
q_0	$\{q_0, q_3\}$	$\{q_0, q_1\}$
q_1	ϕ	$\{q_2\}$
q_2	$\{q_2\}$	$\{q_2\}$
q_3	$\{q_4\}$	ϕ
q_4	$\{q_4\}$	$\{q_4\}$