INSTITUTO TECNOLOGICO DE IZTAPALAPA

MATERIA: Lenguajes y Autómatas II

PROFESOR: Abiel Tomás Parra Hernández

ALUMNO:

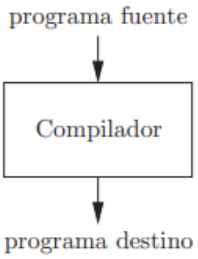
*COFRADIA RODRIGEZ RODRIGO B. 161080399

CARRERA: SISTEMAS COMPUTACIONALES

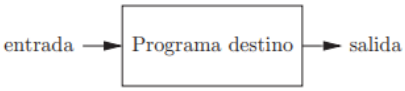
Apuntes individuales

1.1 Procesadores de lenguaje

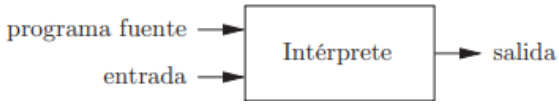
Dicho en forma simple, un compilador es un programa que puede leer un programa en un lenguaje (el lenguaje fuente) y traducirlo en un programa equivalente en otro lenguaje (el lenguaje destino). Una función importante del compilador es reportar cualquier error en el programa fuente que detecte durante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

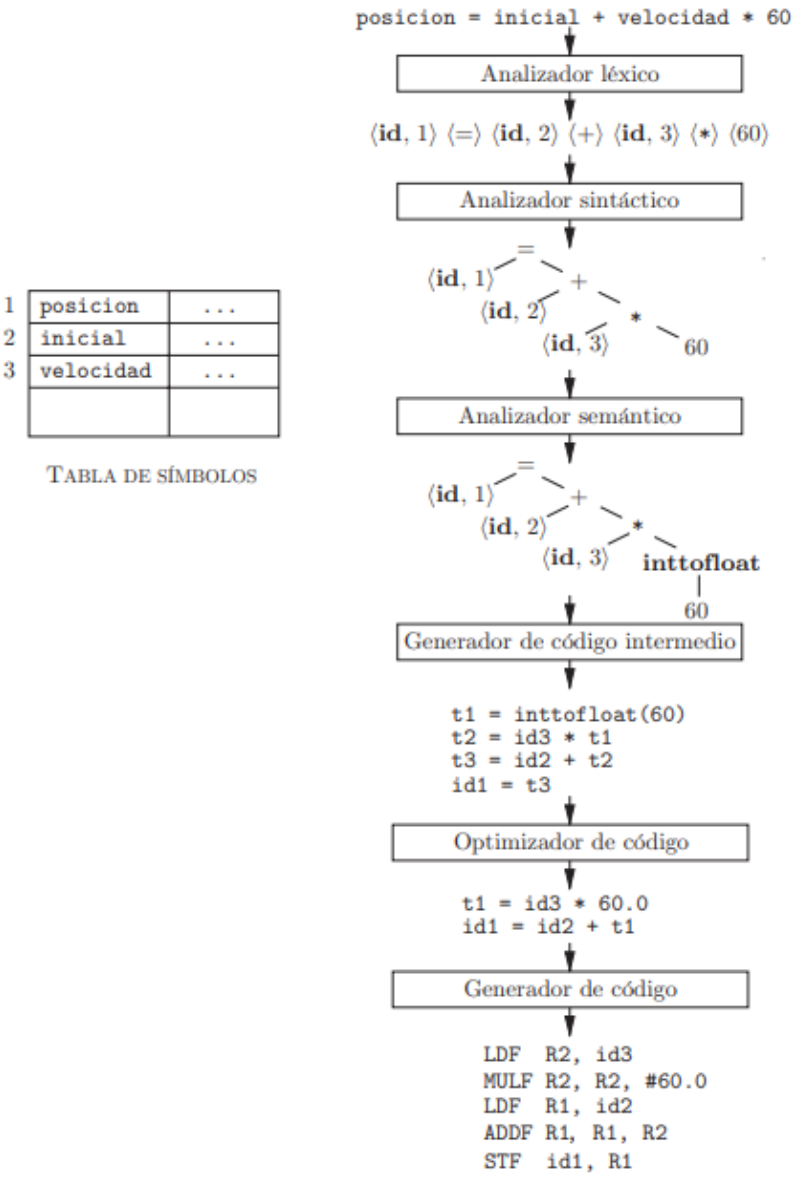
1.2.1 Análisis de léxico

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

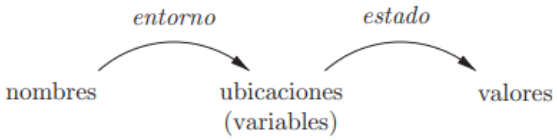
En esta sección hablaremos sobre la terminología más importante y las distinciones que aparecen en el estudio de los lenguajes de programación. No es nuestro objetivo abarcar todos los conceptos o todos los lenguajes de programación populares. Asumimos que el lector está familiarizado por lo menos con 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.

Apuntes semana 5

Las propiedades teóricas se mantienen incluso en el contexto de las lenguas así que estamos interesados en algunas otras propiedades con respecto a las operaciones recién introducidas concatenación, cierre de kleene y cierre positivo. Encuentra esa concatenación de lenguajes asociativos ya lo hemos mostrado lo que si es que L1, L2, L3 están de la mando para poder resolver o acceder a la operación lo cual L1, L2, L3 se leen en serie no se puede hacer lo de

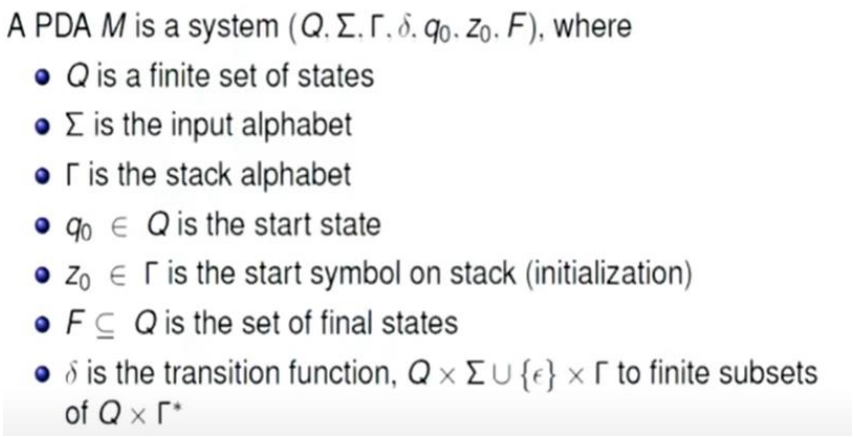
L3, L2, L1 se puede tener una variedad de caracteres los cuales serán válidos ya que sirven para un sinfín de cadenas en la elaboración de las operaciones, las gramática es una familia la cual es un conjunto de reglas la cual se usa para construir o validar frases de lenguajes, los símbolos tienen que ser terminales.

Apuntes semana 6

En la actividad de esta semana se hicieron las anotaciones de los videos "Syntax Analysis: Context-free Grammars, Pushdown Automata and Parsing" y estos nos explican de la gramática libre de contextos.

Pushdown: Es un conjunto finito de estados Q, tiene un alfabeto de entrada tiene una coma de alfabeto de pila hay un estado de pila de inicio z y un conjunto de estados finales F.

En el conjunto Q de un estado q en un símbolo de entrada y los símbolos de pila agregan en la parte superior de stack pueden moverse al estado P1 o P2 o es p1,p3 y en ese proceso quita la parte superior del símbolo de la pila y lo reemplaza con gama1, gama2 cualquiera de estos dependiendo de en que estado se mueva este es el símbolo de entrada por uno.



Este tiene una forma gramatical que se utiliza para los idiomas libres de contexto, este pesa bastante entonces requiere 1 cuadro de tiempo para que así estén dos algoritmos muy conocidos.

Algoritmo de Earley

El es un algoritmo no determinista de análisis sintáctico para las gramáticas libres de contexto descrito originalmente por el informático estadounidense Jay Earley en 1970. Se ordena, a los lados de los algoritmos CYK y GLR, entre los algoritmos que usan la noción de reparto

(de cálculos y de estructuras) y que construyen todos los análisis posibles de una frase. Es uno de los algoritmos no deterministas que usan ideas de la programación dinámica.

Algoritmo de Cocke-Younger-Kasami (CYK)

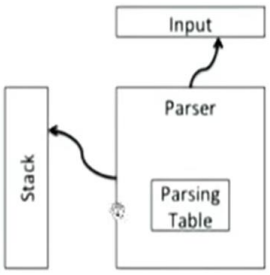
Este algoritmo es el que determina si una cadena puede ser generada por una gramática libre de contexto y, si es posible, cómo puede ser generada. Este proceso es conocido como análisis sintáctico de la cadena. El algoritmo es un ejemplo de programación dinámica.

Los subconjuntos de lenguajes libres de contexto suelen requerir 0(n) tiempo

* Análisis predictivo utilizando gramática LL(1)(análisis de método de arriba hacia abajo).

* Reduzca el análisis sintáctico utilizando la gramática LA(1) (método de análisis sintáctico de abajo hacia arriba).

El análisis de arriba hacia abajo usando pases predictivos eleva la derivación más a la izquierda de la cadena mientras se construyen las tres primeras, así que comenzamos desde el símbolo de inicio y predecimos la producción que se usa en la derivación, necesitamos más información y eso se conoce como tabla de análisis, que se construye sin conexión y se detiene, por lo que vamos a estudiar cómo la partícula es efectiva en la tabla de análisis que se construye fuera de línea y se almacena. Entonces vamos a estudiar cómo se construye la tabla de análisis sintáctico. Las próximas entrevistas de producción en la derivación se determinan observando el siguiente símbolo y también la tabla del analizador así que esta combinación te dice exactamente qué producción en particular se utilizará y el símbolo X que vemos se llama cuando miramos hacia adelante así que al imponer restricciones a la gramática, nos aseguramos de que no haya luego una reducción en cualquier tipo de tabla de personas, por lo que veremos que si hay más de una producción en cualquier trama de la tabla de cierre, entonces no podemos decidir qué producción usar, en el momento de la construcción de la tabla de análisis, hay dos producciones elegibles para colocarse en el mismo tipo de tabla de personas.




```
Initial configuration: Stack = S, Input = w$,
where, S = start symbol, $ = end of file marker
repeat {
   let X be the top stack symbol;
   let a be the next input symbol /*may be $*/;
   if X is a terminal symbol or $ then
       if X == a then {
           pop X from Stack;
           remove a from input;
       } else ERROR();
   else  /* X is a non-terminal symbol */
       if M[X,a] == X → Y1Y2... Yk then {
           pop X from Stack;
           push Yk, Yk-1, ..., Y1 onto Stack;
                (Y1 on top)
       }
} until Stack has emptied;
```

Condiciones comprobables parta ll(1)

Puede ser un símbolo de terminal o el final de la tabla de símbolos de Entonces lo que tenemos que hacer es entender es que en primer lugar, por qué está determinado sólo por cómo sigue

Se sabe muy simplemente con **First Step** como primero que no es más que el primero que son todas las cadenas de S Prime en realidad derivadas de S Y que si derivan todas las cuerdas que son, ya sabes derivadas por S comienzan con A ó C. Así que primero de s sería una C

Literalmente por el primero de ll y Tienes todas las cadenas derivables de un sabes que comienzan con la pequeña B. Y porque hay una S no termina aquí todas las cadenas que son derivables por S también están incluidas en la primera forma. y para B, entonces eso nos da a saber, el primero de a tiene a b c en este momento en lo que respecta a B, los símbolos son y el primero de los suyos están incluidos en el primero , así es como se completa todo esto.

Semana 7

parte 1

La gramática se usa para describir la sintaxis de lenguajes de programación como por ejemplo si consideramos un lenguaje de programación como C o Pascal una gramática puede ser escrita para describir la estructura sintáctica.
La gramática son subclases del lenguaje de programación.
Un analizador de la gramática de un lenguaje de programación:

- Verifica que la cadena de tokens para un programa en ese idioma se pueda generar a partir de esa gramática
- Informa cualquier error de sintaxis en el programa
- Construye una representación de árbol de análisis del programa
- Por lo general, llama al analizador léxico para proporcionarle una ficha cuando sea necesario.
- Podría ser escrito a mano o generado automáticamente
- Se basa en gramáticas libres de contexto

Las gramáticas son mecanismos generativos como la expresión regular
Los autómatas pushdown son máquinas que reconocen lenguajes libres de contexto como FSA o el autómata finito-estado.
Las gramáticas libres de contexto se denota como
G = N.T.P.S
N = conjunto finito de no terminales
T = conjunto finito de terminales
S e N = El símbolo de inicio
P = Conjunto finito de producciones, y todas las producciones son de la forma
A flecha alfa yendo a alfa
Sólo P es específica y la primera producción corresponde a la del símbolo de inicio

Parte 2 videos
El lenguaje generado por esa gramática se denota como L de G, pero como para las expresiones regulares, escribimos L de R aquí escribimos L de G, este es un conjunto de cadenas w. W es un conjunto de cadenas, por lo que no tiene no terminales, por lo que T start es un conjunto de terminales y T start es su cierre. Entonces w es el cierre de T start, eso significa que es una cadena de símbolos terminales, y acabamos de describir el proceso de derivación, entonces, S deriva w, S es un símbolo de inicio, por lo que todas esas cadenas, que pueden derivarse del símbolo de inicio y pertenecen al conjunto de cadenas terminal estrella, es un cierre de la cadena terminal.

El PDA en el estado con el símbolo de entrada a y el símbolo de la parte superior de la pila z, puede ingresar cualquier parte del estado p1, reemplazar el símbolo z por la cadena y avanzar el cabezal de entrada en un símbolo.

No determinístico y Determinístico PDA

Parte 3
Será el nuevo símbolo de la parte superior de la pila y entonces también definiremos la aceptación del lenguaje por un autómata de empuje hacia abajo, uno es por estado final y el otro es por pila vacía. Para la aceptación por estado final, la máquina debe comenzar desde el estado de inicio, pasar a algún estado final y vaciar la entrada también y la pila no importa. Para la aceptación por pila vacía, comienza desde las películas de estado de inicio a algún estado, pero en el proceso no sólo vacía la entrada sino también la pila. Entonces, el estado

al que se filma no es muy importante y, por lo tanto, a veces lo establecemos F igual a phi para este tipo de un autómata.

Los autómatas de empuje hacia abajo son tan iguales que en el caso de los autómatas de estado finito no deterministas, tenemos autómatas de empuje hacia abajo no deterministas y similar a DFA, tenemos DPDA. Sin embargo, en el caso de NFA y DFA se demostró que eran equivalentes; en otras palabras, el lenguaje que fue aceptado por NFA es también el lenguaje aceptado por cualquier DFA equivalente. Entonces podemos convertir cada NFA en un DFA donde en el caso de NPDA, NPDA es estrictamente más potente que la clase DPDA

Parsing

El análisis (parsing) es el proceso de construir un árbol de análisis sintáctico para una oración generada por una gramática determinada

si no hay restricciones sobre el idioma y la forma de gramática utilizada, los analizadores de lenguajes libres de contexto requieren un tiempo de cubo O (n) para el análisis, por lo que hay dos algoritmos muy conocidos:

• Todo lenguaje posee una serie de reglas para describir los programas fuentes (sintaxis).

• Un analizador sintáctico implementa estas reglas haciendo uso de GICs Gramáticas

• Son un formalismo matemático que permite decidir si una cadena pertenece a un lenguaje dado. • Se define como la cuarteta G= (N, Σ, S, P), en donde N es el conjunto de símbolos terminales, Σ es conjunto de símbolos terminales, S es el símbolo inicial (S pertenece a N) y P es un conjunto de reglas de producción.

Gramáticas

• Los símbolos no terminales (N) son aquellos que pueden seguir derivando en otros; mientras que los terminales el proceso finaliza allí.

• Las reglas de producción siguen el formato: αβ donde α y β pertenecen a N y Σ en cualquier forma.

Reglas de producción

• Son las reglas que permiten decidir si la cadena pertenece a un lenguaje y la estructura que lleva:

• S A|aB B ε

• A aA|bC C ε

• S Genera cadenas del lenguaje a*b u a

Tipos de gramáticas

• Las gramáticas más sencillas son las gramáticas regulares, debido a que no presentan anomalías de ningún tipo. Desafortunadamente este tipo de gramáticas no permiten expresar todos los lenguajes posibles y en especial los humanos por lo que se necesitan otros tipos de gramáticas. Las más utilizadas en informáticas son las libres del contexto.

Tipo de gramática que acepta un analizador sintáctico Nosotros nos centraremos en el análisis sintáctico para lenguajes basados en gramáticas formales, ya que de otra forma se hace muy difícil la comprensión del compilador, y se pueden corregir, quizás más fácilmente, errores de muy difícil localización, como es la ambigüedad en el reconocimiento de ciertas sentencias.

Parte 4 videos

Análisis sintáctico por descenso recursivo
Se puede considerar el análisis sintáctico descendente como un intento de encontrar una derivación por la izquierda para una cadena de entrada También se puede considerar como un intento de construir un árbol de análisis sintáctico para la entrada comenzando desde la raíz y creando los nodos del árbol en orden previo.
El descenso recursivo, puede incluir retrocesos, es decir, varios exámenes de la entrada. Sin embargo, no hay muchos analizadores sintácticos con retroceso. En parte, porque casi nunca se necesita el retroceso para analizar sintácticamente las construcciones de los lenguajes de programación. En casos como el análisis sintáctico del lenguaje natural, el retroceso tampoco es muy eficiente, y se prefieren los métodos tabulares, como el algoritmo de programación dinámica o el método de Earley

Parte 5

El análisis sintáctico LR es un método de análisis sintáctico ascendente y es sinónimo de exploración de izquierda a derecha con la derivación más a la derecha en reversa, siendo k el número de tokens de anticipación, es importante porque se pueden generar de forma automática utilizando generadores de La gramática LR libre de contexto es un subconjunto de gramáticas libres de contexto, para las cuales se pueden construir tales analizadores.

La configuración tiene dos partes, una es la pila, la otra es la entrada sin gastar o sin usar y para comenzar con la pila como solo el símbolo de inicio o el estado inicial del analizador sintáctico y la entrada no expandida como la entrada completa, terminó con un final de archivo o una marca de dólar.
Las dos partes en la tabla de parsing son: **Action y Goto**
La tabla de acciones puede tener cuatro tipos de cambio de entrada, reducir, aceptar y error
La tabla Goto se utiliza para proporcionar la siguiente información del estado, que es realmente necesaria después de un movimiento de reducción.

Gramática LR

Se dice que una gramática es LR (k) si para cualquier cadena de entrada dada, en cada paso de cualquier derivación más a la derecha, se puede detectar el identificador beta examinando la cadena phi beta y escaneando como máximo, los primeros k símbolos de la cadena de entrada no utilizada t.

Parte 6

Si un estado contiene un elemento de la forma [A a.] ("Reducir el articulo "), entonces una reducción de la producción A --àa es el acción en ese estado si no hay "elementos reducidos" en un estado, entonces shift es el acción apropiada

Podría haber conflictos de cambio-reducir o reducir-reducir conflictos en un estado

Ambos elementos de desplazamiento y reducción están presentes en el mismo estado (Conflicto S-R)

Hay más de un elemento de reducción en un estado (R-R conflicto)

Es normal tener más de un elemento de turno en un estado (no los conflictos de turno-turno son posibles) o Si no hay conflictos S-R o R-R en cualquier estado de un LR (0) DFA, entonces la gramática es LR (0), de lo contrario, no es LR (0)NPTEL

Seguir (S) = ($) donde $ es EOF

Reducción en S y cambia a + y, resolverá los conflictos Esto es similar a tener un marcador de final como #

Si la gramática no es LR (0), intentamos resolver conflictos en los estados usando un símbolo de anticipación

Parte 7

Los analizadores sintácticos LR (1) tienen una gran cantidad de estados para C, muchos miles de estados

Un analizador SLR (1) (o LR (0) DFA) para C tendrá algunos cien estados (con muchos conflictos)

Los analizadores LALR (1) tienen exactamente el mismo número de estados como analizadores SLR (1) para la misma gramática, y se derivan de analizadores LR (1)

Los analizadores SLR (1) pueden tener muchos conflictos, pero LALR (1) los analizadores pueden tener muy pocos conflictos

Si el analizador LR (1) no tuvo conflictos S-R, entonces el analizador LALR (1) derivado correspondiente tampoco tendrá ninguno

Sin embargo, esto no es cierto con respecto a los conflictos R-R

Los analizadores LALR (1) son tan compactos como los analizadores SLR (1) y son casi tan potentes como los analizadores sintácticos LR (1)

Uno de los estados s1 ', a partir del cual se genera s1, debe tienen los mismos elementos básicos que s1 el elemento [A .. a está en s1 ', entonces s1' también debe tener el elemento [B 3.a.c (la búsqueda anticipada no necesita ser b en s1 '- puede ser b en algún otro estado, pero eso no es de interés para algunos.

Estos dos elementos en s1 'todavía crean un conflicto S-R en el analizador LR (1) Por lo tanto, la fusión de estados con un núcleo común nunca puede introducir un nuevo conflicto S-R, porque el cambio depende sólo en el núcleo, no en el futuro sin embargo, la fusión de estados puede introducir una nueva R-R conflicto en el analizador LALR (1) a pesar de que el original

El analizador LR (1) no tenía ninguno

Estas gramáticas son raras en la práctica. Aquí hay uno del libro de ALSU. Por favor construya los juegos completos de elementos LR (1) como trabajo a domicilio:

S S $. S aAd bBd | aBe bAe

A— c. B - c

Dos estados contienen los elementos:

{[A → c .. d]. [B → Cm el} y

{[A C .. e. [B C. d]}

La fusión de estos dos estados produce el estado LALR (1):

{{A → C .. d e]. [B → C .. d el}

Este estado LALR (1) tiene un conflicto de reducir-reducir

El escritor del compilador identifica las principales no terminales, como los de programa, declaración, bloque, expresión, etc.

Agrega a la gramática, producciones de error de la forma un error donde A es un no terminal mayor y a es un cadena adecuada de símbolos gramaticales (generalmente terminal símbolos), posiblemente vacío

Gramática de Tributo parte 1

La consistencia semántica que no se puede manejar en la etapa de análisis se maneja aquí. Por ejemplo, los analizadores no pueden manejar características sensibles al contexto de los lenguajes de programación. por lo que hay dos tipos de semántica que tienen los lenguajes de programación, uno se conoce como **semántica estática** y el otro se conoce como **semántica dinámica.**
Semántica estática: como su nombre indica, usted sabe que no depende del sistema de tiempo de ejecución y la ejecución del programa, sino que depende únicamente de la definición del lenguaje de programación.
Semántica dinámica: nuevamente, como el nombre lo indica, son propiedades de los sistemas de lenguaje de programación que ocurren en tiempos de ejecución, y necesitamos verificar tales propiedades sólo durante el tiempo de ejecución del programa.
Por ejemplo, la semántica estática puede ser verificada por un analizador semántico o usted sabe que hay muchos ejemplos aquí, por lo que todas las variables se declaran antes de su uso, si es así, todo es de lo contrario se debe proporcionar un mensaje de error. Luego, haga coincidir los tipos de la expresión y la variable a la que está asignada en los dos lados de una declaración de asignación y haga coincidir los tipos de parámetros y el número de parámetros tanto en la declaración como en el uso.

Las gramáticas de atributos son extensiones de la gramática libre de contexto. Así que sea G igual a N T P S una gramática libre de contexto, así que la base es definitivamente una gramática libre de contexto y deja que el conjunto de variables V de la gramática sea N unión T.Para cada símbolo de X de V podemos asociar un conjunto de atributos denotados como X punto a X punto b, etc., que Es por eso que el nombre atributo gramatical. Hay dos tipos de atributos **heredados** que se denotan como Al de X, por lo que los atributos inherentes de X y los atributos **sintetizados** que se denotan como AS de X son realmente conjuntos de atributos. Cada atributo toma valores de un dominio específico, podría ser un dominio finito o podría ser un dominio infinito y llamamos al dominio que conoce tal dominio como sus tipos

1. Un atributo no se puede sintetizar y heredar, pero un símbolo puede tener ambos tipos de atributos
2. Los atributos de los símbolos se evalúan sobre un árbol de análisis sintáctico marcando pasadas sobre el árbol de análisis
3. Los atributos sintetizados se calculan de abajo hacia arriba desde las hojas hacia arriba
4. Los atributos heredados fluyen desde el padre o los hermanos hasta el nodo en cuestión

Gráfico de dependencia de atributos

Parte 2

Dos tipos de atributos: heredados y sintetizados. Cada atributo toma valores de un dominio específico. Una producción p e P tiene un conjunto de reglas de cálculo de atributos para

Atributos sintetizados del LHS no terminal de p, atributos heredados de los no terminales RHS de p

Las reglas son estrictamente locales para la producción p (sin efectos secundarios)

Un atributo no se puede sintetizar y heredar al mismo tiempo, pero un símbolo puede tener ambos tipos de atributos

Los atributos de los símbolos se evalúan en un árbol de análisis por haciendo pases sobre el árbol de análisis

Los atributos sintetizados se calculan de abajo hacia arriba. Moda de las hojas hacia arriba

Siempre sintetizado a partir de los valores de los atributos de los niños. Los nodos hoja (terminales) tienen atributos sintetizados (solo) inicializado por el analizador léxico y no se puede modificar

Los atributos heredados fluyen desde el padre o los hermanos el nodo en cuestión

AG para la evaluación de un número real a partir de su cadena de bits representación

Ejemplo: 110.101 = 6.625

• N + L.R. L- BL | B. R BR | B, B 0 | 1

• AS (N) = AS (R) = AS (B) = {valor ↑: real}.

AS (L) = {longitud ↑: entero, valor ↑: real}

N L.R {N.valor ↑: = L.valor ↑ -R.valor t}

B (L.valor t: = B.value ↑: L.longth t: = 1}

+ BL2 {L1.longitud t = L2.longitud t +1:

2->L1.valor = B.valor t 2-2. longitud + L2. valor t}

R B (Rvalue = B.valor ↑ / 2}

R BR2 (R valor t = (B.valor t + R2.valor t) / 2)}

OB0 (B.valor t = 0}

B 1 {B.valuet: -1}

Parte 3

Cada símbolo X de V tiene asociado un conjunto de atributos, Dos tipos de atributos: heredados y sintetizados, Cada atributo toma valores de un dominio específico

Un pEP de producción tiene un conjunto de reglas de cálculo de atributos para

Atributos sintetizados del LHS no terminal de p

Atributos heredados de los no terminales RHS de p

Las reglas son estrictamente locales para la producción p (sin efectos secundarios)

Un AG con solo atributos sintetizados es un S-atribuido gramática. Los atributos de SAGS se pueden evaluar en cualquier orden ascendente sobre un árbol de análisis (paso único). La evaluación de atributos se puede combinar con el análisis LR(YACC)

En las gramáticas con atributos L, las dependencias de atributos siempre ir de izquierda a derecha

Más precisamente, cada atributo debe ser

Sintetizado o Heredado, pero con las siguientes limitaciones:

Considere una producción p: A X, X2..Xp. Sea X.a e AI (X;).

Xj.a solo puede usar o elementos de Al (A)

Elementos de A / (Xx) o AS (X). k = 1 ... i-1 (es decir, atributos de X..X-1)

Nos concentramos en SAGS y LAGS de 1 paso, en los que

La evaluación de atributos se puede combinar con LR. LL o RD analizando

Entrada: un árbol de análisis T con instancias de atributos no evaluados

Salida: T con valores de atributo consistentes

void dfvisit (n: nodo)

{para cada hijo m de n, de izquierda a derecha hacer

{evaluar atributos heredados de m;

dfvisit (m)

}:

Evaluar atributos sintetizados de n

Evaluación de atributos no se puede hacer a lo largo con análisis LR

| El análisis de LL no es posible.

La gramática es recursiva a la izquierda

Un AG para asociar información de tipo con nombres en declaraciones de variables

• Al (L) = Al (ID) = {tipo: {integer.real}}

AS (T) = {tipo ↑: {integer.real}}

AS (ID) = AS (identificador) = {nombre ↑: cadena}

DListD DList; re

D L: T {L.type: = T.type ↑}

OT int {Tiype t = integer}

OT flotante (T.tipo t: = real)}

ID {ID.type = L.type}

OLL2. ID (tipo L2: = L1.type 1: ID.type: = L1.type |}

3.2.2 "Gramáticas independientes del contexto (Context-Free Grammars)"

La solución tradicional de gramática libre de contexto es utilizar una gramática libre de contexto (cfg).La solución tradicional de gramática libre de contexto es utilizar una gramática libre de contexto (cfg).
Una gramática libre de contexto, G, es un conjunto de reglas que describen cómo formar las expresiones senSentence. La colección de oraciones que se pueden derivar de G se llama una cadena de símbolos que se puede derivar de las reglas de un lenguaje gramatical definido por G, denotado G. El conjunto de lenguajes definidos por gramáticas libres de contexto se denomina conjunto de lenguajes libres de contexto

GRAMÁTICAS SIN CONTEXTO

Una gramática libre de contexto G es cuádruple (T, NT, S, P) donde:
T es el conjunto de símbolos terminales, o palabras, en el lenguaje L (G). Los símbolos de terminal corresponden a categorías sintácticas devueltas por el escáner.

NT es el conjunto de símbolos no terminales que aparecen en las producciones de G. Los no terminales son variables sintácticas introducidas para proporcionar abstracción y estructura en las producciones.

Los conjuntos T y NT pueden derivarse directamente del conjunto de producciones, P. El símbolo de inicio puede no ser ambiguo, como en la gramática SheepNoise, o puede no ser obvio, como en la siguiente gramática:
Par → (bracket) bracket→ [Par]
| ( ) | [ ]
En este caso, la elección del símbolo de inicio determina la forma de los corchetes exteriores. Usar Paren como S asegura que cada oración tenga un par de paréntesis más externo, mientras que usar Bracket fuerza un par de cuadrados más externo soportes.

FORMULARIO BACKUS-NAUR

La notación tradicional utilizada por los científicos informáticos para representar una gramática libre de contexto se llama forma Backus-Naur o BNF. BNF denota símbolos no terminales envolviendolos entre corchetes angulares, como {SheepNoise}. Los símbolos terminales estaban subrayados. El símbolo :: = significa "deriva" y el símbolo | significa "también deriva". En BNF, la gramática del ruido de oveja se convierte en:

{SheepNoise} :: = baa {SheepNoise}
| baa
Esto es completamente equivalente a nuestra gramática SN.

Para aplicar reglas en SN para derivar oraciones en L (SN) debemos identificar el símbolo de meta o símbolo de inicio de SN. El símbolo de objetivo representa el conjunto de todas las cadenas de L (SN). Debe ser uno de los símbolos no terminales introducidos para agregar estructura y abstracción al lenguaje. Dado que SN tiene solo un no terminal, SheepNoise debe ser el objetivo símbolo.

Para derivar una oración, comenzamos con una cadena de prototipo que contiene solo el símbolo del objetivo, SheepNoise. Elegimos un símbolo no terminal, α, en la cadena del prototipo, elegimos una regla gramatical, α → β, y reescribimos α con β. Repetimos esto proceso de reescritura hasta que la cadena prototipo no contenga más no terminales, momento en el que se compone enteramente de palabras o símbolos terminales y es una oración en el idioma.
Ejemplo Complejo:

Comenzando con el símbolo de inicio, Expr, podemos generar dos tipos de subterráneos: subterráneos entre paréntesis, con la regla 1, o subterráneos simples, con la regla 2. Para generar la oración "(a + b) × c", podemos usar la siguiente reescritura secuencia (2,6,1,2,4,3), que se muestra a la izquierda. Recuerda que la gramática se ocupa de categorías sintácticas, como el nombre en lugar de lexemas como a, b o c.

Este cfg simple para expresiones no puede generar una oración con paréntesis desequilibrados o incorrectamente anidados. Solo la regla 1 puede generar un paréntesis abierto; también genera el paréntesis de cierre coincidente. Por lo tanto, no puede generar cadenas como "a + (b × c" o "a + b) × c)" y un analizador creado a partir de la gramática no aceptará tales cadenas.

La derivación de (a + b) × c reescribió, en cada paso, quedando más a la derecha una derivación que reescribe, en cada paso, el símbolo no terminal no termina más a la derecha. Este comportamiento sistemático fue una elección; otras opciones son posibles. Una alternativa obvia es reescribir el no terminal más a la izquierda en cada paso. El uso de las opciones más a la izquierda produciría una secuencia de derivación derivada diferente para la misma oración. La derivación más a la izquierda de (a + b) × c una derivación que reescribe, en cada paso, la no terminal más a la izquierda sería:

4.2 Y 4.2.2 UNA INTRODUCCIÓN A LOS SISTEMAS DE TIPO

Los lenguajes de programación asocian una colección de propiedades con cada valor de datos. Llamamos a esta colección de propiedades el tipo de valor.
El tipo especifica un conjunto de propiedades que comparten todos los valores de ese tipo. Los tipos se pueden especificar por membresía; por ejemplo, un número entero podría ser cualquier número entero i en el rango $-2\ 31 \leq i < 2\ 31$.

En un tipo de colores enumerados, definido como el conjunto {rojo, naranja, amarillo, verde, azul, marrón, negro, blanco}. Los tipos se pueden especificar mediante reglas; por ejemplo, la declaración de una estructura en c define un tipo. En este caso, el tipo incluye cualquier objeto con los campos declarados en el orden declarado; los campos individuales tienen tipos que especifican los rangos de valores permitidos y su interpretación. (Representamos el tipo de estructura como el producto de los tipos de sus campos constituyentes, en orden). Algunos tipos están predefinidos por un lenguaje de programación; otros son construidos por el programador. El conjunto de tipos en un lenguaje de programación, junto con las reglas que utilizan tipos para especificar el comportamiento del programa, se denominan colectivamente sistema de tipos.

4.2.2 Componentes de un sistema de tipos pt

Un sistema de tipos para un lenguaje moderno típico tiene cuatro componentes principales: un conjunto de tipos básicos o tipos integrados; reglas para construir nuevos tipos a partir de los tipos existentes; un método para determinar si dos tipos son equivalentes o compatibles; y reglas para inferir el tipo de cada expresión del idioma de origen. Muchos lenguajes también incluyen reglas para la conversión implícita de valores de un tipo a otro según el contexto.

- Números

Los lenguajes exponen la implementación de hardware subyacente mediante la creación de distintos tipos para diferentes implementaciones de hardware. Por ejemplo, c, c ++ y Java distinguen entre enteros con signo y sin signo.
fortran, pl / iyc exponen el tamaño de los números. Tanto cy fortran especifican la longitud de los elementos de datos en términos relativos. Por ejemplo, un doble en fortran tiene el doble de longitud que un real. Sin embargo, ambos lenguajes le dan al compilador control sobre la longitud de la categoría de número más pequeña.
Por el contrario, las declaraciones pl / i especifican una longitud en bits. El compilador mapea esta longitud deseada en una de las representaciones de hardware. Por lo tanto, la implementación de ibm 370 de pl / i asignó una variable binaria fija (12) y una variable binaria fija (15) a un entero de 16 bits, mientras que un binario fijo (31) se convirtió en un entero de 32 bits.

El lenguaje define una jerarquía de tipos de números, pero permite al implementador seleccionar un subconjunto para soportar. Sin embargo, el estándar establece una distinción cuidadosa entre números exactos e inexactos y especifica un conjunto de operaciones que deben devolver un número exacto cuando todos sus argumentos son exactos. Esto proporciona un grado de flexibilidad al implementador, al tiempo que le permite al programador razonar sobre cuándo y dónde puede ocurrir la aproximación.

- Caracteres

un carácter es una sola letra. Durante años, debido al tamaño limitado de los alfabetos occidentales, esto llevó a una representación de un solo byte (8 bits) para los caracteres, generalmente mapeados en el
conjunto de caracteres ascii. Recientemente, más implementaciones, tanto del sistema operativo como del lenguaje de programación, han comenzado a admitir conjuntos de caracteres más grandes expresados en el formato estándar Unicode, que requiere 16 bits. La mayoría de los lenguajes asumen que el juego de caracteres está ordenado, por lo que los operadores de comparación estándar, como <, = y>, funcionan de forma intuitiva, imponiendo el orden lexicográfico. La conversión entre un carácter y un número entero aparece en algunos idiomas.

- Booleanos

La mayoría de los lenguajes de programación incluyen un tipo booleano que toma dos valores: verdadero y falso. Las operaciones estándar proporcionadas para los valores booleanos incluyen y, o, xor y no. Los valores booleanos, o expresiones con valores booleanos, se utilizan a menudo para determinar el flujo de control. c considera los valores booleanos como un subrango de los enteros sin signo, restringidos a los valores cero (falso) y uno (verdadero).

- Tipos compuestos y construidos

Los tipos básicos de un lenguaje de programación proporcionan una abstracción adecuada de los tipos reales de datos manejados directamente por el hardware, a menudo son inadecuados para representar el dominio de información que necesitan los programas.

Los programas tratan habitualmente con estructuras de datos más complejas, como gráficos, árboles, tablas, matrices, registros, listas y pilas. Estas estructuras constan de uno o más objetos, cada uno con su propio tipo. La capacidad de construir nuevos tipos para estos objetos compuestos o agregados es una característica esencial de muchos lenguajes de programación.

- Matrices

Las matrices se encuentran entre los objetos agregados más utilizados. Una matriz agrupa varios objetos del mismo tipo y le da a cada uno un nombre distinto, aunque sea un nombre calculado implícito en lugar de un nombre explícito designado por el programador. La declaración c int a [100 [200]; reserva espacio para 100 × 200 = 20.000 enteros y se asegura de que se puedan direccionar con el nombre a. Las referencias a [1] [17] y a [2] [30] acceden a ubicaciones de memoria distintas e independientes. La propiedad esencial de una matriz es que el programa puede calcular nombres para cada uno de sus elementos usando números (o algún otro tipo discreto ordenado) como subíndices.

Una matriz de enteros de 10 × 10 tiene un tipo de matriz bidimensional de enteros. Algunos idiomas incluyen las dimensiones de la matriz en su tipo; por tanto, una matriz de enteros de 10 × 10 tiene un tipo diferente que una matriz de enteros de 12 × 12. Esto permite que el compilador capture operaciones de matriz en las que las dimensiones son incompatibles como un error de tipo. La mayoría de los lenguajes permiten matrices de cualquier tipo base; algunos lenguajes también permiten matrices de tipos construidos.

- Instrumentos de cuerda

Algunos lenguajes de programación tratan las cadenas como un tipo construido. pl / i, por ejemplo, tiene cadenas de bits y cadenas de caracteres. Las propiedades, atributos y operaciones definidas en ambos tipos son similares; son propiedades de una cadena. El rango de valores permitido en cualquier posición difiere entre una cadena de bits y una cadena de caracteres. Por lo tanto, es apropiado verlos como una cadena de bits y una cadena de caracteres. (La mayoría de los lenguajes que admiten cadenas limitan la compatibilidad incorporada a un solo tipo de cadena: la cadena de caracteres). Otros idiomas, como c, admiten cadenas de caracteres manipulándolas como matrices de caracteres.

- Tipos enumerados

Muchos lenguajes permiten al programador crear un tipo que contiene un conjunto específico de valores constantes. Un tipo enumerado, introducido en Pascal, permite al programador utilizar nombres autodocumentados para pequeños conjuntos de constantes. Los ejemplos clásicos incluyen días de la semana y meses. En la sintaxis c, estos podrían ser:
enum WeekDay {Monday, Tuesday, Wednesday,
Thursday, Friday, Saturday, Sunday};
enum Month {January, February, March, April, May, June, July, August, September,

October, November, December};
El compilador asigna cada elemento de un tipo enumerado a un valor distinto.
Los elementos de un tipo enumerado están ordenados, por lo que las comparaciones entre elementos del mismo tipo tienen sentido.

- Estructuras y variantes

Las estructuras, o registros, agrupan varios objetos de tipo arbitrario. Los elementos, o miembros, de la estructura suelen recibir nombres explícitos.
Por ejemplo, un programador que implemente un árbol de análisis sintáctico en c podría necesitar nodos con uno y dos hijos.

5.1 TAXONOMÍA DE REPRESENTACIONES INTERMEDIAS

Los compiladores han utilizado muchos tipos de IR. Organizaremos nuestra discusión sobre el IRS en tres ejes: organización estructural, nivel de abstracción y disciplina de denominación. En general, estos tres atributos son independientes

Las 3 categorías estructurales:
■ Los IR gráfico: codifican el conocimiento del compilador en un gráfico. Los algoritmos se expresan en términos de objetos gráficos: nodos, bordes, listas o árboles.
■ Los IR lineales: se asemejan al pseudocódigo de alguna máquina abstracta. Los algoritmos iteran sobre secuencias de operaciones simples y lineales. El códigoILOC utilizado en este libro es una forma de ir lineal.
■ Los IR híbridos: combinan elementos de los IR gráficos y lineales, en un intento de capturar sus puntos fuertes y evitar sus debilidades. Una representación híbrida común utiliza un ir lineal de bajo nivel para representar bloques de código de línea recta y un gráfico para representar el flujo de control entre esos bloques.

IRS GRÁFICO

Muchos compiladores usan irs que representan el código subyacente como un gráfico. Si bien todos los IR gráficos constan de nodos y bordes, difieren en su nivel de abstracción, en la relación entre el gráfico y el código subyacente y en la estructura del gráfico.

Árboles relacionados con la sintaxis

Los árboles de parsé son una forma específica de árboles arbóreos. En la mayoría de los árboles arbóreos, la estructura del árbol corresponde a la sintaxis del código fuente.

Analizar árboles

La figura muestra la gramática de expresión clásica junto a un árbol de análisis sintáctico para a × 2 + a × 2 × b. El árbol de análisis es grande en relación con el texto de origen porque representa la derivación completa, con un nodo para cada símbolo gramatical en la derivación. Dado que el compilador debe asignar memoria para cada nodo y cada borde, y

debe atravesar todos esos nodos y bordes durante la compilación, vale la pena considerar formas de reducir este árbol de análisis.

Árboles de sintaxis abstracta
El árbol de sintaxis abstracta (ast) conserva la estructura esencial del árbol de análisis pero elimina los nodos extraños. La precedencia y el significado de la expresión permanecen, pero los nodos extraños han desaparecido. Aquí está el ast para a × 2 + a × 2 × b:

Nivel de abstracción

Las técnicas basadas en árboles para la optimización y la generación de código, de hecho, pueden requerir tal detalle. Como ejemplo, considere el enunciado w ← a - 2 × b. Un ast a nivel de fuente crea un formulario conciso. Sin embargo, el árbol a nivel de fuente carece de muchos de los detalles necesarios para traducir la declaración en código ensamblador. Un árbol de bajo nivel, puede hacer explícito ese detalle. Este árbol presenta cuatro nuevos tipos de nodos. Un nodo val representa un valor que ya está en un registro. Un nodo núm representa una constante conocida. Un nodo de laboratorio representa una etiqueta de nivel de ensamblaje, generalmente un símbolo reubicable. Finalmente, es un operador que desreferencia un valor; trata el valor como una dirección de memoria y devuelve el contenido de la memoria en esa dirección.

Gráficos

Mientras que los árboles proporcionan una representación natural de la estructura gramatical del código fuente descubierto por análisis, su estructura rígida los hace menos útiles para representar otras propiedades de los programas. Para modelar estos aspectos del comportamiento del programa, los compiladores suelen utilizar gráficos más generales como irs. El dag presentado en la sección anterior es un ejemplo de gráfico.

Gráfico de dependencia

Los compiladores también usan gráficos para codificar el flujo de valores desde el punto donde se crea un gráfico de dependencia de datos un gráfico que modela el flujo de valores desde las definiciones hasta los usos en un fragmento de código, se crea un valor, una definición, hasta cualquier punto donde se usa, un uso.

IRS LINEALES
Un programa en lenguaje ensamblador es una forma de código lineal. Consiste en una secuencia de instrucciones que se ejecutan en su orden de aparición (o en un orden consistente con ese orden). Las instrucciones pueden contener más de una operación; si es así, esas operaciones se ejecutan en paralelo. Los ir lineales utilizados en los compiladores se parecen al código ensamblador de una máquina abstracta.

Los ir lineales imponen un orden claro y útil en la secuencia de operaciones.

Si se utiliza un ir lineal como representación definitiva en un compilador, debe incluir un mecanismo para codificar transferencias de control entre puntos del programa. El flujo de control en un ir lineal generalmente modela la implementación del flujo de control en la máquina objetivo. Por lo tanto, los códigos lineales generalmente incluyen saltos y saltos condicionales. El flujo de control delimita los bloques básicos en un ir lineal; los bloques terminan en las ramas, en los saltos o justo antes de las operaciones etiquetadas

Tipos de ir lineales.
■ Los códigos de una dirección modelan el comportamiento de las máquinas acumuladoras y las máquinas apiladoras. Estos códigos exponen el uso de nombres implícitos por parte de la máquina para que el compilador pueda adaptar el código para ellos.
■ Los códigos de dos direcciones modelan una máquina que tiene operaciones destructivas. Estos códigos cayeron en desuso a medida que las limitaciones de la memoria se volvieron menos importantes; un código de tres direcciones puede modelar operaciones destructivas de forma explícita.
■ Los códigos de tres direcciones modelan una máquina donde la mayoría de las operaciones toman dos operandos y producen un resultado.

6.1 INTRODUCTION PROCEDURE ABSTRACTION

El proceso es una de las abstracciones centrales en la mayoría de los lenguajes de programación modernos. El programa crea un entorno de ejecución controlado; cada proceso tiene su propio almacenamiento con nombre dedicado. Los procedimientos ayudan a definir las interfaces entre los componentes del sistema; las interacciones entre los componentes generalmente se construyen a través de llamadas a procedimientos.

La última característica (a menudo llamada compilación separada) nos permite crear grandes sistemas de software. Si el compilador necesita el texto completo del programa para cada compilación, los grandes sistemas de software no serán sostenibles.

¡Imagínese que cada cambio de edición realizado durante el proceso de desarrollo recompilará una aplicación multimillonaria!

Por lo tanto, el proceso juega un papel vital en el diseño y la ingeniería del sistema, al igual que en el diseño del lenguaje y la implementación del compilador.

Este capítulo presenta las técnicas utilizadas para implementar procedimientos y llamadas a procedimientos. Específicamente, verifica la implementación de controles, haming e interfaces de llamada. Estas abstracciones encapsulan muchas de las características que hacen que los lenguajes de programación estén disponibles y permiten la construcción de sistemas a gran escala.

Visión general

Este proceso es una de las abstracciones centrales de la mayoría de los lenguajes de programación. El proceso crea un entorno de ejecución controlado. Cada proceso tiene su propio almacenamiento con nombre dedicado. Las declaraciones ejecutadas en el proceso pueden acceder a variables privadas o locales en el almacenamiento privado

El proceso juega un papel importante en el proceso de desarrollo del programador del software y del programa de traducción del compilador. Tres abstracciones clave

El programa proporcionado permite la construcción de programas no triviales.

1. Resumen de llamada a procedimiento El lenguaje de procedimiento admite la abstracción de llamada a procedimiento. Cada idioma tiene un mecanismo estándar para llamar a un procedimiento y asignar un conjunto de parámetros o parámetros desde el espacio de nombres de la persona que llama al espacio de nombres de la persona que llama.

Esta abstracción generalmente implica devolver el control a

2. Espacio de nombres En la mayoría de los idiomas, cada proceso crea un nuevo espacio de nombres protegido. Los programadores pueden declarar nuevos nombres, como el parámetro Current