República Bolivariana de Venezuela
Ministerio del Poder Popular para la Educación
Universidad José Antonio Páez
Facultad de Ingeniería

TRADUCTORES E INTERPRETADORES.

Nombre:

Andreas Peralta 26.389.791

San Diego, 01 de julio del 2022

Sintaxis:

Se presenta la gramática o gramática de libre contexto, que se utiliza para especificar la sintaxis o esquema de un lenguaje Una gramática describe en forma natural la estructura jerárquica de la mayoría de las instrucciones de un lenguaje de programación.

Árboles:

Un árbol de análisis sintáctico muestra, en forma gráfica, la manera en que el símbolo inicial de una gramática deriva a una cadena en el lenguaje.

Las estructuras de datos tipo árbol figuran de manera prominente en la compilación.

• Un árbol consiste en uno o más nodos. Los nodos pueden tener etiqueta. Al dibujar un árbol, con frecuencia representamos los nodos mediante estas etiquetas solamente.

• Sólo uno de los nodos es la raíz. Todos los nodos, excepto la raíz, tienen un padre único; la raíz no tiene padre. Al dibujar árboles, colocamos el padre de un nodo encima de ese nodo y dibujamos una línea entre ellos. Entonces, la raíz es el nodo más alto (superior).

• Si el nodo N es el padre del nodo M, entonces M es hijo de N. Los hijos de nuestro nodo se llaman hermanos. Tienen un orden, partiendo desde la izquierda, por lo que al dibujar árboles, ordenamos los hijos de un nodo dado en esta forma.

• Un nodo sin hijos se llama hoja. Los otros nodos (los que tienen uno o más hijos) son nodos interiores.

• Un descendiente de un nodo N es ya sea el mismo N, un hijo de N, un hijo de un hijo de N, y así en lo sucesivo, para cualquier número de niveles. Decimos que el nodo N es un ancestro del nodo M, si M es descendiente de N.

Ambigüedad:

Una gramática puede tener más de un árbol de análisis sintáctico que genere una cadena dada de terminales. Se dice que dicha gramática es ambigua donde debemos buscar una cadena de terminales que sea la derivación de más de un árbol de análisis sintáctico.

Como una cadena con más de un árbol de análisis sintáctico tiene, por lo general, más de un significado, debemos diseñar gramáticas no ambiguas para las aplicaciones de compilación, o utilizar gramáticas ambiguas con reglas adicionales para resolver las ambigüedades.

Asociatividad de operadores:

En la mayoría de los lenguajes de programación, los cuatro operadores aritméticos (suma, resta, multiplicación y división) son asociativos por la izquierda. Algunos operadores comunes, como la exponenciación, son asociativos por la derecha. Como otro ejemplo, el operador de asignación = en C y sus descendientes es asociativo por la derecha; es decir, la expresión a=b=c se trata de la misma forma que la expresión a=(b=c).

Precedencia de operadores:

Las reglas de asociatividad para + y * se aplican a las ocurrencias del mismo operador, por lo que no resuelven esta ambigüedad. Las reglas que definen la precedencia relativa de los operadores son necesarias cuando hay más de un tipo de operador presente. Decimos que * tiene mayor precedencia que +, si * recibe sus operandos antes que +. En la aritmética ordinaria, la multiplicación y la división tienen mayor precedencia que la suma y la resta.

Traducción dirigida por la sintaxis:

Se realiza uniendo reglas o fragmentos de un programa a las producciones en una gramática.

En esta sección se introducen dos conceptos relacionados con la traducción orientada a la sintaxis:

• Atributos. Un atributo es cualquier cantidad asociada con una construcción de programación. Algunos ejemplos de atributos son los tipos de datos de las expresiones, el número de instrucciones en el código generado, o la ubicación de la primera instrucción en el código generado para una construcción, entre muchas otras posibilidades. Cómo utilizamos símbolos de la gramática (no terminales y terminales) para representar las construcciones de programación, extendemos la noción de los atributos, de las construcciones a los símbolos que las representan.

• Esquemas de traducción (dirigida a la sintaxis). Un esquema de traducción es una notación para unir los fragmentos de un programa a las producciones de una gramática. Los fragmentos del programa se ejecutan cuando se utiliza la producción durante el análisis sintáctico. El resultado combinado de todas estas ejecuciones de los fragmentos, en el orden inducido por el análisis sintáctico, produce la traducción del programa al cual se aplica este proceso de análisis/síntesis.

Análisis sintáctico descendente recursivo:

Es posible que un analizador sintáctico de descenso recursivo entre en un ciclo infinito. Se produce un problema con las producciones “recursivas por la izquierda”.

en donde el símbolo más a la izquierda del cuerpo es el mismo que el no terminal en el encabezado de la producción.

Se puede eliminar una producción recursiva por la izquierda, reescribiendo la producción problemática

Traductor de expresiones:

A menudo, la gramática subyacente de un esquema dado tiene que modificarse antes de poder analizarlo con un analizador sintáctico predictivo.

Un punto inicial útil para diseñar un traductor es una estructura de datos llamada árbol sintáctico abstracto. En un árbol sintáctico abstracto para una expresión, cada nodo interior representa a un operador; los hijos del nodo representan a los operandos del operador. De manera más general, cualquier construcción de programación puede manejarse al formar un operador para la construcción y tratar como operandos a los componentes con significado semántico de esa construcción.

Análisis Léxico:

Un analizador léxico lee los caracteres de la entrada y los agrupa en “objetos token”. Junto con un símbolo de terminal que se utiliza para las decisiones de análisis sintáctico, un objeto token lleva información adicional en forma de valores de atributos. Hasta ahora, no hemos tenido la necesidad de diferenciar entre los términos “token” y “terminal”, ya que el analizador sintáctico ignora los valores de los atributos que lleva un token. En esta sección, un token es un terminal junto con información adicional. A una secuencia de caracteres de entrada que conforman un solo token se le conoce como lexema. Por ende, podemos decir que el analizador léxico aísla a un analizador sintáctico de la representación tipo lexema de los tokens. El analizador léxico en esta sección permite que aparezcan números, identificadores y “espacio en blanco” (espacios, tabuladores y nuevas líneas) dentro de las expresiones. Puede usarse para extender el traductor de expresiones.

Tablas de Símbolos:

Las tablas de símbolos son estructuras de datos que utilizan los compiladores para guardar información acerca de las construcciones de un programa fuente. La información se recolecta en forma incremental mediante las fases de análisis de un compilador, y las fases de síntesis la utilizan para generar el código destino. Las entradas en la tabla de símbolos contienen información acerca de un identificador, como su cadena de caracteres (o lexema), su tipo, su posición en el espacio de almacenamiento, y cualquier otra información relevante. Por lo general, las tablas de

símbolos necesitan soportar varias declaraciones del mismo identificador dentro de un programa.

El analizador léxico, el analizador sintáctico y el analizador semántico son los que crean y utilizan las entradas en la tabla de símbolos durante la fase de análisis.

En algunos casos, un analizador léxico puede crear una entrada en la tabla de símbolos, tan pronto como ve los caracteres que conforman un lexema. Más a menudo, el analizador léxico sólo puede devolver un token al analizador sintáctico, por decir id, junto con un apuntador al lexema. Sin embargo, sólo el analizador sintáctico puede decidir si debe utilizar una entrada en la tabla de símbolos que se haya creado antes, o si debe crear una entrada nueva para el identificador.

*Tabla de símbolos por alcance: El término "alcance del identificador x" en realidad se refiere al alcance de una declaración específica de x. El término alcance por sí solo se refiere a una parte del programa que es el alcance de una o más declaraciones. Los alcances son importantes, ya que el mismo identificador puede declararse para distintos fines en distintas partes de un programa. A menudo, los nombres comunes como i y x tienen varios usos. Como otro ejemplo, las subclases pueden volver a declarar el nombre de un método para redefinirlo de una superclase.

*las tablas de símbolos para los bloque: Las implementaciones de las tablas de símbolos para los bloques pueden aprovechar la regla del bloque anidado más cercano. El anidamiento asegura que la cadena de tablas de símbolos aplicables forme una pila. En la parte superior de la pila se encuentra la tabla para el bloque actual. Debajo de ella en la pila están las tablas para los bloques circundantes. Por ende, las tablas de símbolos pueden asignarse y desasignarse en forma parecida a una pila. Algunos compiladores mantienen una sola hash table de entradas accesibles; es decir, de entradas que no se ocultan mediante una declaración en un bloque anidado. Dicha hash table soporta búsquedas esenciales en tiempos constantes, a expensas de insertar y eliminar entradas al entrar y salir de los bloques.

En efecto, la función de una tabla de símbolos es pasar información de las declaraciones a los usos. Una acción semántica "coloca" (put) información acerca de un identificador x en la tabla de símbolos, cuando se analiza la declaración de x. Posteriormente, una acción semántica asociada con una producción como factor → id "obtiene" (get) información acerca del identificador, de la tabla de símbolos. Como la traducción de una expresión E1 op E2, para un operador op ordinario, depende sólo de las traducciones de E1 y E2, y no directamente de la tabla de símbolos, podemos agregar cualquier número de operadores sin necesidad de cambiar el flujo básico de información de las declaraciones a los usos, a través de la tabla de símbolos.

Máquinas de pilas abstractas:

La arquitectura del conjunto de instrucciones de la máquina destino tiene un impacto considerable en la dificultad de construir un buen generador de código que produzca código máquina de alta calidad. Las arquitecturas más comunes de las máquinas de destino son RISC (reduced instruction set computer), CISC (complex instruction set

computer) y basadas en pilas. Por lo general, una máquina RISC tiene muchos registros, instrucciones de tres direcciones, modos de direccionamiento simple y una arquitectura del conjunto de instrucciones relativamente sencilla. En contraste, una máquina CISC, por lo general, tiene menos registros, instrucciones de dos direcciones, una variedad de modos de direccionamiento, varias clases de registros, instrucciones de longitud variable e instrucciones con efectos adicionales. En una máquina basada en pila, las operaciones se realizan metiendo operandos en una pila, y después llevando a cabo las operaciones con los operandos en la parte superior de la pila. Para lograr un alto rendimiento, la parte superior de la pila se mantiene, por lo general, en los registros. Estas máquinas desaparecieron casi por completo, ya que se creía que la organización de la pila era demasiado limitante y requería demasiadas operaciones de intercambio y copiado. No obstante, las arquitecturas basadas en pila revivieron con la introducción de la Máquina virtual de Java (JVM). Ésta es un intérprete de software para bytecodes de Java, un lenguaje intermedio producido por los compiladores de Java. El intérprete proporciona compatibilidad de software a través de varias plataformas, un importante factor en el éxito de Java. 8.1 Cuestiones sobre el diseño de un generador de código 507 Generación de código Para superar el castigo sobre el alto rendimiento de la interpretación, que puede estar en el orden de un factor de 10, se crearon los compiladores Java just-in-time (JIT). Estos compiladores JIT traducen los códigos de byte en tiempo de ejecución al conjunto de instrucciones de hardware nativo de la máquina de destino. Otro método para mejorar el rendimiento de Java es construir un compilador que compile directamente en las instrucciones de máquina de la máquina destino, pasando por alto los bytecodes de Java por completo. Producir un programa en lenguaje máquina absoluto como salida tiene la ventaja de que puede colocarse en una ubicación fija en la memoria, y ejecutarse de inmediato. Los programas pueden compilarse y ejecutarse con rapidez. Producir un programa en lenguaje máquina reubicable (que a menudo se le conoce como módulo objeto) como salida permite que los subprogramas se compilen por separado. Un conjunto de módulos objeto reubicables puede enlazarse y cargarse para que lo ejecute un cargador de enlace. Aunque debemos sufrir la sobrecarga adicional de enlazar y cargar si producimos módulos objeto reubicables, obtenemos mucha flexibilidad al poder compilar las subrutinas por separado y llamar a otros programas (compilados con anterioridad) desde un módulo objeto. Si la máquina destino no maneja la reubicación de manera automática, el compilador debe proporcionar información de reubicación explícita al cargador para enlazar los módulos del programa que se compilaron por separado. Producir un programa en lenguaje ensamblador como salida facilita de alguna manera el proceso de la generación de código. Podemos generar instrucciones simbólicas y utilizar las facilidades de las macros del ensamblador para ayudar en la generación de código. El precio que se paga es el paso de ensamblado después de la generación de código. En este capítulo utilizaremos una computadora tipo RISC muy simple como nuestra máquina destino. Le agregaremos algunos modos de direccionamiento tipo CISC, para poder hablar también sobre las técnicas de generación de código para las máquinas CISC. Por cuestión de legibilidad, usaremos código ensamblador como el lenguaje destino. Siempre y cuando puedan calcularse direcciones a partir de desplazamientos y que la demás información se almacene en la tabla de símbolos, el generador de código puede producir direcciones reubicables o absolutas para los nombres, con la misma facilidad que las direcciones simbólicas.