

## **INTEGRANTES:**

**Villalobos Romero Katherin 171080154**

**Hernandez Hernandez Diego 171080165**

**Gomez Rubio Israel 171080093**

**Cofradía Rodríguez Rodrigo 161080399**

## **LIBRO COMPILADORES**

En este capítulo se presentan las distintas formas de los traductores de lenguaje, proporcionan una descripción general de alto nivel sobre la estructura de un compilador ordinario, y hablan sobre las tendencias en los lenguajes de programación y la arquitectura de máquinas que dan forma a los compiladores. Incluyen algunas observaciones sobre la relación entre el diseño de los compiladores y la teoría de las ciencias computacionales, y un esquema de las aplicaciones de tecnología sobre los compiladores que van más allá de la compilación. Hablan acerca de los conceptos clave de los lenguajes de programación que necesitaremos para nuestro estudio de los compiladores. Los lenguajes de programación son anotaciones que describen los cálculos a las personas y las máquinas. Nuestra percepción del mundo en que vivimos depende de los lenguajes de programación, ya que todo el software que se ejecuta en todas las computadoras se escribió en algún lenguaje de programación. Pero antes de poder ejecutar un programa, primero debe traducirse a un formato en el que una computadora pueda ejecutarlo.

Los sistemas de software que se encargan de esta traducción se llaman compiladores.

Este libro trata acerca de cómo diseñar e implementar compiladores. Aquí descubriremos que podemos utilizar unas cuantas ideas básicas para construir traductores para una amplia variedad de lenguajes y máquinas. Además de los compiladores, los principios y las técnicas para su diseño se pueden aplicar a tantos dominios distintos aparte, que es probable que un científico computacional las reutilice muchas veces en el transcurso de su carrera profesional.

El estudio de la escritura de los compiladores se relaciona con los lenguajes de programación, la arquitectura de las máquinas, la teoría de lenguajes, los algoritmos y la ingeniería de software.

El parser utiliza los primeros componentes de los tokens producidos por el analizador de léxico para crear una representación intermedia en forma de árbol que describa la estructura gramatical del flujo de tokens. Una representación típica es el árbol sintáctico, en el cual cada nodo interior representa una operación y los hijos del nodo representan los argumentos de la operación

Una parte importante del análisis semántico es la comprobación de tipos, en donde el compilador verifica que cada operador tiene operandos que coincidan. 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.

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. 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

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 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.

Estos atributos pueden proporcionar información acerca del espacio de almacenamiento que se asigna para un nombre, su tipo, su alcance, 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 y el tipo devuelto.

### La analogía entre rango estático y dinámico

Aunque puede haber cualquier número de directivas de alcance estático o dinámico, las reglas de alcance estáticas normales están estructuradas por bloques e instrucciones dinámicas regulares. En cierto sentido, las reglas dinámicas son para la regla estática del tiempo tiene que ver con el espacio. Aunque las reglas estáticas

Se requiere encontrar la oración cuya unidad (bloque) está más cerca de la ubicación física utilizada, y la regla dinámica requiere que encontremos la oración cuya unidad (bloque) está más cerca (La llamada del procedimiento) es lo más cercano al tiempo de uso.

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

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

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

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

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

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

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

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

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

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

## VIDEO DE ANALISIS LEXICO

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

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

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

### **Función del Analizador Léxico**

Su principal función consiste en leer los caracteres de entrada y elaborar como salida una secuencia de componentes léxicos que utiliza el analizador sintáctico para hacer el análisis. En la imagen se puede apreciar el esquema de una interacción que se aplica convirtiendo el analizador léxico en una subrutina o corrutina del analizador sintáctico. Recibida la orden "obtener el siguiente componente léxico" del analizador sintáctico, el analizador léxico lee los caracteres de entrada hasta que pueda identificar el siguiente componente léxico.

### **Funciones secundarias.**

Como el analizador léxico es la parte del compilador que lee el texto fuente. También puede realizar ciertas funciones secundarias en la interfaz del usuario, como eliminar del programa fuente comentarios y espacios en blanco en forma de caracteres de espacio en blanco, caracteres TAB y de línea nueva. Otra función es relacionar los mensajes de error del compilador con el programa fuente. Por ejemplo, el analizador léxico puede tener localizado

el número de caracteres de nueva línea detectados, de modo que se pueda asociar un número de línea con un mensaje de error.

En algunos compiladores, el analizador léxico se encarga de hacer una copia del programa fuente en el que están marcados los mensajes de error. Si el lenguaje fuente es la base de algunas funciones de pre procesamiento de macros, entonces esas funciones del preprocesador también se pueden aplicar al hacer el análisis léxico.

El análisis Léxico es la primera fase de un compilador, y toma el programa fuente de los pre-procesadores que está escrito en forma de declaraciones. Este proceso entonces desglosa el código en una serie de tokens, deshaciéndose primero de todos los comentarios en el código y los espacios en blanco.

Si el analizador léxico encuentra un token que no es válido (ya hablamos de esto cuando describimos las fases del compilador) entonces se genera un error. El analizador léxico trabaja muy de cerca con el analizador sintáctico. Leemos caracteres de entrada a través del código fuente, revisamos que los tokens sean todos válidos y pasamos los lexemas al analizador sintáctico cuando este lo requiera.

## **Tokens**

Los lexemas son una secuencia de caracteres alfanuméricos en un token. Hay varias reglas predefinidas para cada lexema que sirven para identificarlos como un token válido. Estas reglas están definidas por gramáticas formales, que siguen un patrón, un patrón explica que es un token, y estos patrones pueden ser definidos a través de expresiones regulares.

En un lenguaje de programación, palabras reservadas, constantes, identificadores, cadenas de texto, números, operadores y signos de puntuación pueden ser considerados como tokens.

Consideremos la siguiente expresión

```
int valor = 100;
```

1

```
int valor = 100;
```

Al pasarla por un analizador léxico el mismo reconocerá cada uno de los tokens en la expresión.

```
int (palabra reservada) valor (identificador) = (operador) 100 (número/constante) ;(símbolo)
```

1

```
int (palabra reservada) valor (identificador) = (operador) 100 (número/constante) ;(símbolo)
```

## Especificaciones de un token

Hay algunos términos en la teoría de lenguajes formales que definiremos a continuación para entender mejor qué es un token.

### Alfabeto o Vocabulario

Es un conjunto finito de símbolos formalmente definidos en un lenguaje. por ejemplo:

$\{0,1\}$  Es un conjunto de símbolos para representar números binarios

$\{0,1,2,3,4,5,6,7,8,9,A,B,C,D,E,F\}$  Es un conjunto de símbolos para representar números hexadecimales

$\{a-z, A-Z\}$  Es un conjunto de símbolos para representar palabras en idioma español o inglés (según sea el caso).

### Cadenas o Strings

Una cadena es una secuencia finita de alfabetos cuya longitud es el número de ocurrencias de un símbolo perteneciente a esos alfabetos, por ejemplo la longitud de "compiladores" es 12 y estaría denotada como  $|compiladores| = 12$ . Una cadena con longitud cero es conocida como una cadena vacía y está denotada por la letra griega epsilon  $\epsilon$ .

## VIDEO DE ANALISIS LEXICO PARTE 2

Los NFAS son FSA que permiten 0, 1 o más transiciones de un estado en un símbolo de entrada dado. Un NFA es un 5-tuplo como antes, pero la función de transición es diferente.  $(q, a) = p$  es el conjunto de todos los estados  $p$ , de modo que hay una transición etiquetada  $a$  de  $q$  a  $p$ . Una cadena es aceptada por un NFA si existe una secuencia de transiciones correspondiente a la cadena, que lleva del estado inicial a algún estado final. Cada NFA puede convertirse a un FA determinístico equivalente (DFA), que acepta el mismo lenguaje que el NFA

$L = \{x \mid x \text{ contiene dos ceros consecutivos o dos unos consecutivos}\}$

El estado inicial de la DFA corresponde al conjunto  $\{q_0\}$  y estará representado por  $q_0$ . A partir de  $S = \{q_0\}$ , los nuevos estados de la DFA se construyen a petición. Cada subconjunto de estados de NFA es un posible estado del DFA. Todos los estados del DFA que contienen algún estado final como miembro. Nfa es equivalente a nfa en poder

Una definición regular es una secuencia de "ecuaciones" de la forma  $d_1 = r_1; d_2 = r_2; \dots; d_n = r_n$ , donde cada  $d_i$  es un nombre distinto, y cada  $r_i$  es una expresión regular sobre los símbolos  $\{U \cup \{d_1, d_2, \dots, d_{i-1}\}\}$

- Identificadores y enteros

Letra =  $a + b + c + d + e$ ; dígito =  $0 + 1 + 2 + 3 + 4$ ; letra de identificación (letra + dígito)\*;  
número = dígito dígito

- Números sin signo

Dígito =  $0 + 1 + 2 + 3 + 4 + 5 + 6 + 7 + 8 + 9$ ;

Dígitos = dígito dígito ";

Fracción opcional = dígitos + €;

Exponente opcional = (E (+ | - |) dígitos) + E

Número sin firmar = Dígitos fracción opcional exponente opcional NPTEL

Sea  $r$  un RE. Entonces existe un NFA con transiciones electrónicas que acepta  $L(r)$ . La prueba es por construcción. Si  $L$  es aceptado por un DFA, entonces  $L$  es generado por un RE. La prueba es tediosa.

Los diagramas de transición son DFAS generalizados con las siguientes diferencias. Los bordes pueden etiquetarse con un símbolo, un conjunto de símbolos o una definición regular. Algunos estados aceptantes pueden indicarse como estados de retracción, indicando que el lexema no incluye el símbolo que nos llevó a un estado aceptante. Cada estado aceptante tiene una acción adjunta a él, que se ejecuta cuando se alcanza ese estado. Normalmente, esta acción devuelve un token y su valor de atributo. Diagramas de transición no están destinados a la máquina, sino sólo a la traducción manual

### VIDEO DE ANALISIS LEXICO PARTE 3

¿Cómo funciona el analizador léxico? Como vimos en el segundo apartado, el analizador léxico funciona bajo demanda del analizador sintáctico cuando le pide el siguiente token. A partir de ese fichero que contiene el código fuente vamos leyendo caracteres que almacenamos en un buffer de entrada. Supongamos el siguiente texto: Si comenzamos donde está la flecha, según accedemos al fichero, tenemos que eliminar dos caracteres en blanco y posicionarnos en  $a$  y leemos ese carácter, devolver ese token, cuando veamos el siguiente espacio en blanco (flecha azul), ver si encaja con alguno de los patrones que hemos definido mediante las expresiones regulares y si es así, pasárselo al analizador sintáctico, que automáticamente nos pedirá que reconozcamos el siguiente token, en este tras eliminar el analizador léxico y el carácter en blanco (donde se encuentra la flecha azul). Estas expresiones regulares se construyen mediante un autómata finito, pero a modo de introducción un autómata finito es un programa que acepta cadenas de un lenguaje definido sobre un alfabeto, y responde si la cadena pertenece al alfabeto o no. Para definir un autómata de una manera formal, necesitamos cinco elementos:

1. alfabeto o conjunto finito de símbolos
2. conjunto finito de estados
3. función de transición de estados
4. estado inicial
5. conjunto de estados finales

Operadores sobre lenguajes

Sean  $L, M$  dos lenguajes  $L \cup M = \{c | c \in L \text{ ó } c \in M\}$

unión de lenguajes  $L \cap M = \{st | s \in L \text{ y } t \in M\}$

concatenación de lenguajes  $LM = \{st | s \in L \text{ y } t \in M\}$

Propuesta: Un analizador léxico para un evaluador de expresiones

- Involucra: – constantes enteras sin signo – operadores relacionales  $<, <=, >, >=, <>, =$  – identificadores de variables

- Componentes léxicos: menor  $\rightarrow <$  mayor  $\rightarrow >$  menor igual  $\rightarrow <=$  mayor igual  $\rightarrow >=$  igual  $\rightarrow =$  distinto  $\rightarrow <>$  letra  $\rightarrow a|b|\dots|z|A|B|\dots|Z$  dígito  $\rightarrow 0|1|\dots|9$  identificador  $\rightarrow \text{letra} (\text{letra} |\text{dígito})^* \text{constEntera} \rightarrow \text{dígito} \text{dígito}^*$

Palabras clave: Palabras con un significado concreto en el lenguaje. Ejemplos de palabras clave en C son while, if, return. . . Cada palabra clave suele corresponder a una categoría léxica.

Habitualmente, las palabras clave son reservadas. Si no lo son, el analizador léxico necesitaría información del sintáctico para resolver la ambigüedad.

Identificadores Nombres de variables, nombres de función, nombres de tipos definidos por el usuario, etc. Ejemplos de identificadores en C son i, x10, valor\_leído. . . Operadores Símbolos que especifican operaciones aritméticas, lógicas, de cadena, etc. Ejemplos de operadores en C son +, \*, /, %, ==, !=, &&. . .

Constantes numéricas Literales 1 que especifican valores numéricos enteros (en base decimal, octal, hexadecimal. . . ), en coma flotante, etc. Ejemplos de constantes numéricas en C son 928, 0xF6A5, 83.3E+2. . .

Constantes de carácter o de cadena Literales que especifican caracteres o cadenas de caracteres. Un ejemplo de literal de cadena en C es "una cadena"; ejemplos de literal de carácter son 'x', '\0'. . .

Símbolos especiales: Separadores, delimitadores, terminadores, etc. Ejemplos de estos símbolos en C son {, }, ;. . . Suelen pertenecer cada uno a una categoría léxica separada.

Hay tres categorías léxicas que son especiales:

Comentarios: Información destinada al lector del programa. El analizador léxico los elimina.

Blancos: En los denominados "lenguajes de formato libre" (C, Pascal, Lisp, etc.) los espacios en blanco, tabuladores y saltos de línea sólo sirven para separar componentes léxicos. En ese caso, el analizador léxico se limita a suprimirlos. En otros lenguajes, como Python, no se pueden eliminar totalmente.

Fin de entrada: Se trata de una categoría ficticia emitida por el analizador léxico para indicar que no queda ningún componente pendiente en la entrada.

## VIDEO DE INTRODUCCIÓN A COMPILADORES

Los principios y técnicas que se usan en la escritura de compiladores se pueden emplear en muchas otras áreas. Se basan en los conceptos de teoría de autómatas y lenguajes formales que se están exponiendo en la parte teórica, y constituyen un campo de aplicación práctica bastante directo.

Su objetivo es, básicamente, traducir un programa (o texto) escrito en un lenguaje “fuente”, que llamaremos programa fuente, en un equivalente en otro lenguaje denominado “objeto”, al que llamaremos programa o código objeto. Si el programa fuente es correcto (pertenece al lenguaje formado por los programas correctos), podrá hacerse tal traducción; si no, se deseara obtener un mensaje de error (o varios) que permita determinar lo más claramente posible los orígenes de la incorrección.

La traducción podrá hacerse en dos formas

interpretación: la traducción se hace “frase a frase”

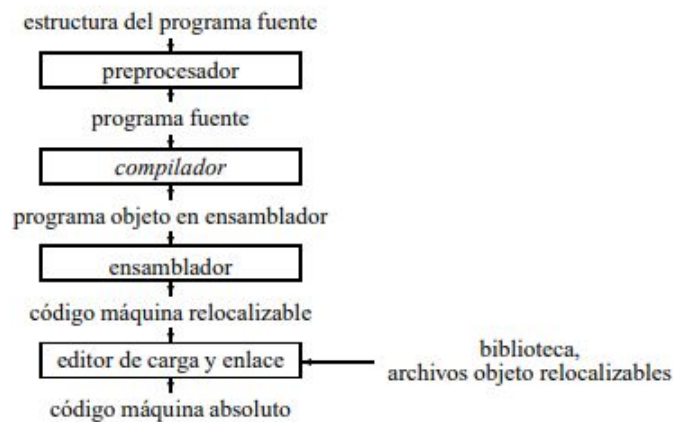
compilación: la traducción se hace del texto completo

Es frecuente que, además del compilador, se utilicen otros programas para crear un código objeto ejecutable. Un esquema típico es el que se describe a continuación. La estructura del programa fuente se escribe, usando algún programa de edición de texto (por ejemplo vi), y puede incluir texto en lenguaje fuente y algunas órdenes para el preprocesador. Este realizará algunas tareas, como eliminación de comentarios, expansión de macros (`#IF . . .`), inclusión de archivos (`#include . . .`), sustitución de constantes (`#define . . .`), o algunas extensiones del lenguaje fuente.

El compilador producirá el resultado del preproceso obteniendo un programa equivalente en lenguaje ensamblador, que a su vez será traducido por el ensamblador a código máquina relocalizable, en el cual las direcciones serán relativas a ciertas posiciones de origen, y quizás algunas llamadas a rutinas no estén resueltas.

Finalmente, el editor de carga y enlace (o “montador”, o “link”) resolverá las llamadas a rutinas, incluyéndose a partir de otros objetos de biblioteca si procede, y obtendrá direcciones absolutas, de modo que ya se dispondrá del código máquina absoluto ejecutable.





No siempre todo este proceso es necesario, y, aún cuando se realice, no siempre será observado por el usuario, de forma que una sola orden (cc programa.c, por ejemplo) puede provocar el proceso completo. Los compiladores suelen incluir opciones que permiten obtener explícitamente (o conservar) los resultados intermedios. Cada uno de los componentes es realmente un traductor. La dificultad máxima de traducción suele estar en la compilación propiamente dicha.

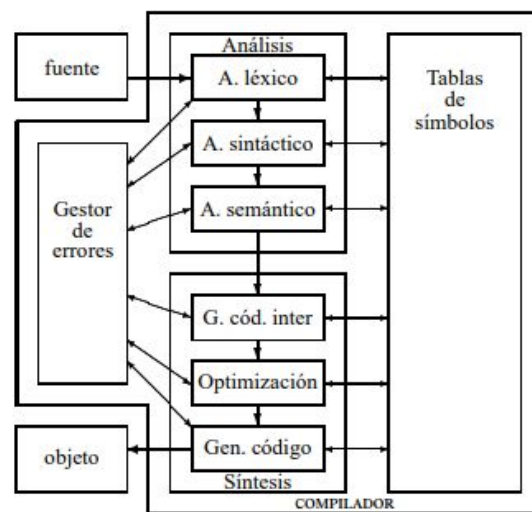
Cuando se ha hablado de programas equivalentes en distintos lenguajes (fuente y objeto), nos referimos a “tener el mismo significado”. Por ello, la compilación requiere dos grandes partes o tareas:

**Análisis** : En la que se analiza el programa fuente para dividirlo en componentes y extraer de algún modo el significado

**Síntesis** : En la que el significado obtenido se escribe en el lenguaje objeto, de las dos, la síntesis es la que requiere técnicas más especializadas.

Durante el análisis, se determinan las operaciones que indica el programa fuente obteniendo una representación del significado, normalmente en una estructura jerárquica, de árbol, en la que cada nodo representa una operación, y cuyos hijos son los argumentos de dicha operación. De este modo, a partir de la representación intermedia, diferentes partes de síntesis podrían obtener distintos códigos, para distintos lenguajes objeto; también, a partir de distintos lenguajes fuente, con partes de análisis adecuadas, se podría obtener una representación intermedia, que una parte de síntesis tradujera a su vez a un único lenguaje objeto. Nuevamente cada una de estas partes es también un traductor. Aún se repite una vez más este esquema de traducciones intermedias en cada una de las partes: el análisis suele dividirse en tres fases: análisis léxico, análisis sintáctico y análisis semántico; y la síntesis en otras tres: generación de código intermedio, optimización de código y generación de código. Se consideran por lo tanto 6 fases en el proceso de compilación, cada una de las cuales transforma la fuente de entrada, progresivamente, hasta conseguir el objeto final.

Por otra parte, cada una de las fases puede detectar errores, y debe informar adecuadamente de ellos. Consideraremos, por simplificar el esquema, un sólo bloque de manipulación de errores, que se usa desde cada fase, puesto que, además, la información de los errores deberá estar relacionada con el lugar del texto en el que se encuentra, relación que sólo puede establecer la fase de análisis léxico. La mayor parte de los errores se detecta en las dos primeras f Además, las diferentes fases crean y acceden a una estructura de datos llamada tabla de símbolos, en la que se anotan a lo largo de las fases variadas informaciones o atributos que es necesario intercambiar; por ejemplo, las variables, con sus nombres, tipos, ámbito, posición relativa de memoria en la generación de código, etc. Estos dos últimos elementos, aunque no sean realmente fases del proceso, tienen suficiente entidad como para ser considerados bloques de similar importancia



Analizaremos algo más las fases usando como ejemplo simple el fragmento de código Pascal posición = inicial + velocidad \* 60

## LIBRO ENGINEERING A COMPILER CHAPTER 1

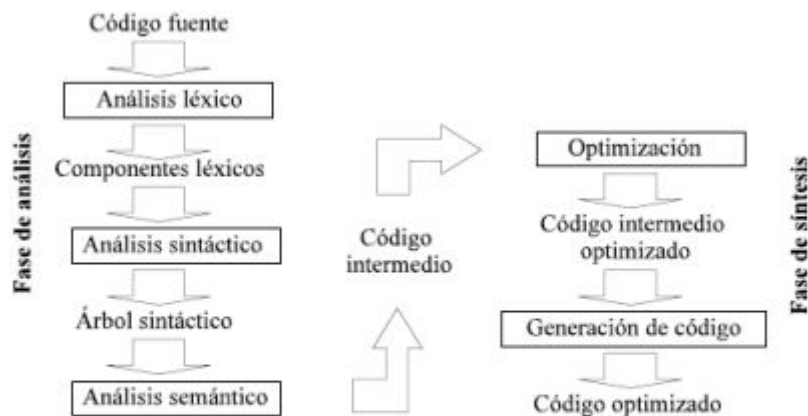
### Overview of Compilation

#### Introduction

Compilación se denomina la fase de codificación en que un programa es traducido del código fuente al código máquina para que pueda ejecutarse. Como tal, la realiza un compilador virtual, cuya tarea consiste en llevar un programa fuente a programa objeto, Un lenguaje de programación está formado por un conjunto de símbolos básicos (alfabeto) y un conjunto de reglas que especifican cómo manipularlos. También debe darle significado a las cadenas formadas al manipular los símbolos básicos.

## Compiler Structure

### Estructura de un Compilador.



*Etapas de traducción de un compilador: análisis y síntesis*

### Análisis léxico.

Un analizador léxico aísla el analizador sintáctico de la representación de lexemas de los componentes léxicos.

El analizador léxico opera bajo petición del analizador sintáctico devolviendo un componente léxico conforme el analizador sintáctico lo va necesitando para avanzar en la gramática. Los componentes léxicos son los símbolos terminales de la gramática.

Suele implementarse como una subrutina del analizador sintáctico. Cuando recibe la orden obtén el siguiente componente léxico, el analizador léxico lee los caracteres de entrada hasta identificar el siguiente componente léxico.

Es la etapa en la que se realiza un análisis a nivel de caracteres. El objetivo de esta fase es reconocer los componentes léxicos presentes en el código fuente, enviándoles después, junto con sus atributos, al analizador sintáctico.

En el estudio de esta fase de análisis léxico es importante que distinguir entre:

Token, el nombre del token es un símbolo abstracto que representa un tipo de unidad léxica. Estos tokens representan palabras reservadas, identificadores, operadores, símbolos especiales, constantes numéricas y caracteres.

Patrón, es una regla que genera la secuencia de caracteres que puede representar a un determinado token.

Lexema, cadena de caracteres que concuerda con un patrón que describe un token. Un token puede tener uno o infinitos lexemas.

La detección de tokens llevada a cabo en esta fase de análisis léxico de un compilador se realiza con gramática y lenguajes regulares.

## **Análisis sintáctico.**

Un analizador sintáctico toma los tokens que le envíe el analizador léxico y creará un árbol sintáctico que refleje la estructura del programa fuente.

En esta fase se comprobará si con dichos tokens se puede formar alguna sentencia válida dentro del lenguaje.

La sintaxis de la mayoría de los lenguajes de programación se define habitualmente por medio de gramáticas libres de contexto. El término libre de contexto se refiere al hecho de que un no terminal puede siempre ser sustituido sin tener en cuenta el contexto en el que aparece.

A partir de estas gramáticas independientes de contexto se diseñan algoritmos de análisis sintáctico capaces de determinar si una determinada cadena de tokens pertenece al lenguaje definido por una gramática dada.

El proceso de comprobación de si una secuencia de tokens pertenece o no a un determinado lenguaje independiente de contexto se lleva a cabo por medio de autómatas a pila.

Los errores detectados en la fase de análisis sintáctico se refieren al hecho de que la estructura que se ha seguido en la construcción de una secuencia de tokens no es la correcta según la gramática que define el lenguaje.

El manejador de errores de un analizador sintáctico debe tener como objetivos:

Indicar y localizar cada uno de los errores encontrados.

Recuperarse del error para poder seguir examinando errores sin necesidad de cortar el proceso de compilación.

No ralentizar en exceso el propio proceso de compilación.

Existen varias estrategias para corregirlos:

En primer lugar se puede ignorar el problema. Esta estrategia consiste en ignorar el resto de la entrada a analizar hasta encontrar un token especial

Otra opción es tratar de realizar una recuperación a nivel de frase, es decir, intentar recuperar el error una vez ha sido detectado.

Otra estrategia en el tratamiento de errores sintácticos es el considerar reglas de producción adicionales para el control de errores.

## **Análisis semántico.**

La semántica se encarga de describir el significado de los símbolos, palabras y frases de un lenguaje, ya sea un lenguaje natural o de programación.

Hay que dotar de significado a lo que se ha realizado en la fase anterior de análisis sintáctico.

Gramáticas atribuidas. Con este tipo de gramáticas se asocia a cada símbolo de la gramática un conjunto de atributos y un conjunto de reglas semánticas. Se conoce como gramática atribuida, o gramática de atributos, a una gramática independiente del contexto

cuyos símbolos terminales y no terminales tienen asociados un conjunto de atributos que representa una propiedad del símbolo.

**Tabla de símbolos.** La tabla de símbolos es una estructura de datos que contiene información por cada identificador detectado en la fase de análisis léxico. Por medio de la tabla de símbolos, el compilador registra los identificadores utilizados en el programa fuente reuniendo información sobre las características de dicho identificador. Estas características pueden proporcionar información necesaria para realizar el análisis semántico.

**Tratamiento de errores.** Durante el análisis semántico el compilador intenta detectar construcciones que tengan estructura sintáctica correcta pero no tengan significado para la operación implicada.

### **Generación de código intermedio.**

En un modelo en el que se realice una separación de fases en análisis y síntesis dentro de un compilador, la etapa inicial traduce un programa fuente a una representación intermedia a partir de la cual se genera después el código objeto.

Entre las ventajas de usar un código intermedio destaca el hecho de que se pueda crear un compilador para una arquitectura distinta sin tocar el front-end de un compilador ya existente para otra arquitectura. De este modo se permite después aplicar un optimizador de código independiente de la máquina a la representación intermedia.

### **Optimización de código intermedio.**

La segunda etapa del proceso de síntesis trata de optimizar el código intermedio, para posteriormente generar código máquina más rápido de ejecutar.

Unos de los tipos de optimización de código más habituales son la eliminación de variables no usadas y el desenrollado de bucles.

También resulta muy habitual traducir las expresiones lógicas para que tenga que calcularse simplemente el valor de aquellos operandos necesarios para poder evaluar la expresión (evaluación en corto circuito).

Otra optimización de código típica es la traducción con precálculo de expresiones constantes.

### **Generación y optimización de código objeto.**

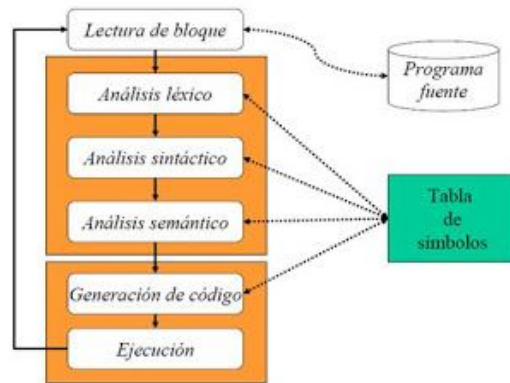
La fase final de un compilador es la generación de código objeto. Cada una de las instrucciones presentes en el código intermedio se debe traducir a una secuencia de instrucciones máquina, donde un aspecto decisivo es la asignación de variables a registros físicos del procesador.

## Overview of Translation

Los intérpretes realizan normalmente dos operaciones:

- Traducen el código fuente a un formato interno.
- Ejecutan o interpretan el programa traducido al formato interno.

La primera parte del intérprete se llama a veces "el compilador", aunque el código interno que genera no es el lenguaje de la máquina, ni siquiera lenguaje simbólico, ni tampoco un lenguaje de alto nivel.



*Particularidades de la interpretación:*

- Ahorra memoria.
- Produce un resultado que no se puede almacenar, lo cual hace la ejecución lenta.
- No demasiado eficiente, cada vez que se entre en un bucle se analizaran sus sentencias.
- Facilita el proceso de depuración.
- No produce resultados transportables

## VIDEO ANÁLISIS SINTÁCTICO PART 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

$S \in N$  = 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

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.

#### Árboles de derivación

- Se puede mostrar como árboles
- Los nodos internos del árbol son todos no terminales y las hojas son todas terminales
- Correspondiente a cada nodo interno  $A$ , existe una producción en  $p$  con el lado derecho de la producción siendo la lista de los hijos de  $A$  leídos de izquierda a derecha
- El rendimiento de un árbol de derivación es la lista de las etiquetas de todas las hojas leídas de izquierda a derecha
- Si alfa es el rendimiento de algún árbol de derivación para la gramática  $G$ , entonces  $S$  es alfa y viceversa

#### Autómatas de empuje (Pushdown Automata)

un PDA  $M$  es un sistema  $(Q, \Sigma, \Gamma, \alpha, q_0, z_0, F)$  donde

$Q$  = es un conjunto finito de estados

$\Sigma$  = es el alfabeto de entrada

$\Gamma$  = es el alfabeto de la pila

$q_0 \in Q$  = es el estado inicial

$z_0 \in \Gamma$  = es el símbolo de inicio en la pila

$F$  = es el conjunto de estados finales

$\Delta$  = es la función de transición

El PDA en el estado que con el símbolo de entrada  $a$  y el símbolo  $z$  de la parte superior de la pila, puede ingresar a cualquiera de los estados  $P$  reemplazar el símbolo  $z$  por la cadena  $w$  y avanzar el encabezado de entrada por un símbolo.

## VIDEO ANÁLISIS SINTÁCTICO PART 2.

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  $p_1$ , reemplazar el símbolo  $z$  por la cadena  $y$  y avanzar el cabezal de entrada en un símbolo.

No determinístico y Determinístico PDA

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  $O(n^3)$  para el análisis, por lo que hay dos algoritmos muy conocidos:

- Cocke-Younger-Kasami's algorithm
- Earley's algorithm

Análisis de arriba hacia abajo usando gramáticas LL

Empieza desde el símbolo de inicio de la gramática y “predicte” la siguiente producción utilizada en la derivación.

Tal “predicción” es ayudada por la tabla de análisis (construida fuera de línea)

la siguiente producción que se utilizará en la derivación se determina utilizando el siguiente símbolo de entrada para buscar la tabla de análisis

en el momento de la construcción de la tabla de análisis, si dos producciones se vuelven elegibles para ser controladas en el mismo espacio de la tabla de análisis, la gramática se declara uniforme para el análisis predictivo.

Gramáticas fuertes de LL (k)



Las gramáticas LL (1) son una subclase de gramáticas libres de contexto, por lo que cuando decimos subclase debemos poner restricciones en la gramática libre de contexto nuevamente, dar un mensaje para verificar si las restricciones se cumplen o no. Así que definamos una clase de gramática llamada gramática LL (k) fuerte y luego veamos cómo se pueden usar para nuestra estrategia de análisis sintáctico LL (1). Entonces, deje que la gramática dada sea G, la entrada se extiende con el símbolo k, por lo que la parte k es en realidad el mirar hacia adelante, por lo que veremos k símbolos en la entrada en un momento particular, por lo que la entrada también debe extenderse por k símbolo de fin de archivo dólar k, k es el adelanto de la gramática. Por tanto, si es LL (1) se nos permite ver exactamente un símbolo en la entrada a la vez, si es LL (2) se nos permite ver dos símbolos a la vez en la entrada y si es LL (k) se nos permite ver k símbolos a la vez en la entrada

Las gramáticas LL (k) fuertes no permiten que se usen diferentes producciones del mismo no terminal incluso en dos derivaciones diferentes, si los primeros k símbolos de las cadenas producidas por la gramática alfa y beta delta son los mismos

### VIDEO DE ANÁLISIS SINTÁCTICO PARTE 3

- 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  $G = (N, \Sigma, S, P)$ , en donde N es el conjunto de símbolos terminales,  $\Sigma$  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:  $\alpha\beta$  donde  $\alpha$  y  $\beta$  pertenecen a N y  $\Sigma$  en cualquier forma.

#### Reglas de producción

- Son las reglas que permiten decidir si la cadena pertenece a un lenguaje y la estructura que lleva:
- $S \rightarrow A|aB \quad B \rightarrow \epsilon$
- $A \rightarrow aA|bC \quad C \rightarrow \epsilon$
- 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 informática 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. La gramática que acepta el analizador sintáctico es una gramática de contexto libre:

- Gramática :  $G (N, T, P, S)$  N = No terminales. T = Terminales. P = Reglas de Producción. S = Axioma Inicial.

## **Árboles de Derivación**

- Es la representación gráfica de la derivación de una cadena
- Se crea utilizando el símbolo inicial como la raíz, los símbolos N representan nodos del árbol y los símbolos  $\Sigma$  las hojas del árbol.
- A través de los árboles de derivación se puede verificar la sintaxis de un lenguaje así como comprobar el significado de las palabras. Árboles de derivación
- Si para la misma cadena existen dos o más árboles de derivación la gramática es ambigua.

## **VIDEO DE ANÁLISIS SINTÁCTICO PARTE 4**

### **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.

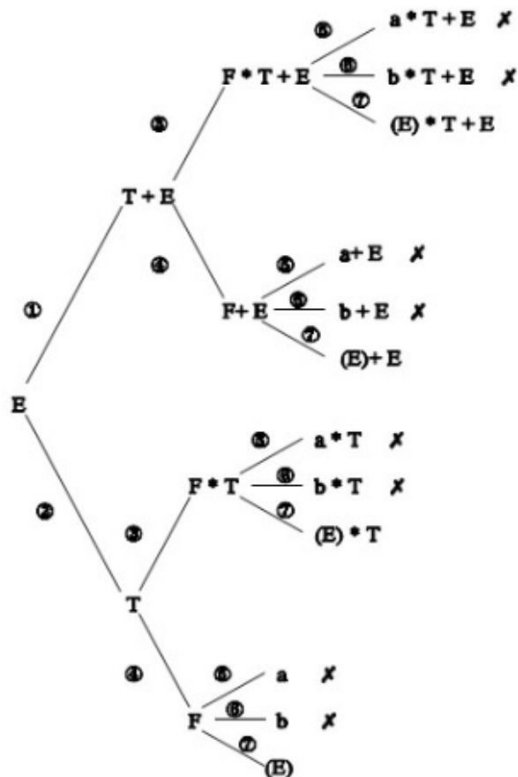
El descenso recursivo, puede incluir retrocesos, es decir, varios exámenes de la entrada. Sin embargo, no hay muchos analizadores sintácticos con retroceso. En parte, porque casi nunca se necesita el retroceso para analizar sintácticamente las construcciones de los lenguajes de programación. En casos 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

Ejemplo Considérese la gramática

$S \rightarrow cAd$

$A \rightarrow ab \mid a$

y la cadena de entrada  $w = cad$ . Para construir un árbol de análisis sintáctico descendente para esta cadena, primero se crea un árbol formado por un solo nodo etiquetado con S. Un apuntador a la entrada apunta a c, el primer símbolo de w. Después se utiliza la primera producción de S para expandir el árbol.



*Algoritmo :*  
 Sea  $k \in \mathbb{N}$  el no terminal más a la izquierda de la forma sentencial.  
 Sea  $\tau \in T^*$  la secuencia de tokens en la izquierda de  $k$ .

```

  Ensayar (forma_sentencial, entrada)
  for  $\{p_i \in P \mid p_i = k \rightarrow \alpha\}$ 
    forma_sentencial' = Aplicar( $p_i$ ,  $k$ , forma_sentencial)
    if  $\tau' = \text{Parte\_izquierda}(\text{entrada})$ 
      if forma_sentencial == entrada
        ¡ Sentencia reconocida !
      else
        Ensayar (forma_sentencial', entrada)
      endif
    endif
  endfor;

```

En el programa principal

```

  Ensayar (S, cadena a reconocer)
  if not ¡ Sentencia reconocida !
    !! Sentencia no reconocida !!
  endif;

```

En todo el árbol de derivaciones, se pretende profundizar por cada rama hasta llegar a encontrar una forma sentencial que no puede coincidir con lo que se busca, en cuyo caso se desecha, o que coincide con lo buscado, momento en que se acepta la sentencia. Si por ninguna rama se puede reconocer, se rechaza la sentencia.

### Analizador descendente

- $L = \{a^n b c^n \mid n > 0\}$
- $S \rightarrow aSc \mid aabc$
- Se requiere una anticipación de por los menos tres símbolos LL(3)
- $S \rightarrow aaAc$
- $A \rightarrow Sc \mid abc$  LL(1)

### Analizador ascendente (LR, LALR)

- Algunos problemas no se pueden resolver de forma descendente ya que no es fácil quitar la ambigüedad. En algunos casos es más fácil demostrar algo ya existente.
- Generalmente los analizadores sintácticos LR(k) son del tipo "bottom-up"

## VIDEO DE ANÁLISIS SINTÁCTICO 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  $t$ .

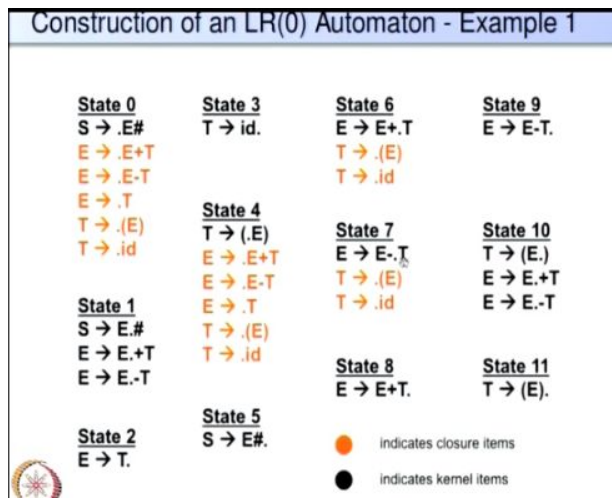
Un prefijo viable de forma oracional  $\phi\beta t$ , donde  $\beta$  denota el identificador, es cualquier prefijo de  $\phi\beta$ . Un prefijo viable no puede contener símbolos a la derecha del identificador.

Los prefijos viables caracterizan los prefijos de formas oracionales que pueden aparecer en la pila de un analizador LR

El DFA de este lenguaje normal puede detectar identificadores durante el análisis de LR

Cuando este DFA alcanza un "estado de reducción", el prefijo viable correspondiente no puede crecer más y, por lo tanto, indica una reducción.

Construimos una gramática aumentada para la que construimos el DFA



## VIDEO DE ANÁLISIS SINTÁCTICO PARTE 6

Si un estado contiene un elemento de la forma  $[A a.]$  ("Reducir el artículo "), entonces una reducción de la producción  $A \rightarrow a$  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

Ejemplo: la expresión gramatical que no es LR (0)

El estado que contiene los elementos  $(T F.)$  y  $(T F. * T)$  tiene conflictos S-R considere el elemento de reducción  $(T F.)$  y los símbolos en SEGUIR (T)

$FOLLOW(T) = \{-. \}$ , \$, Y la reducción por T F puede ser realizado al ver uno de estos símbolos en la entrada (mirar hacia adelante), ya que el cambio requiere ver en la entrada

Recuerde de la definición de  $FOLLOW(T)$  que los símbolos en  $FOLLOW(T)$  son los únicos símbolos que pueden seguir legalmente a  $\rightarrow T$ , en cualquier forma oracional, y por tanto la reducción por T se ve uno de estos símbolos, es correcto

Si los conflictos S-R se pueden resolver usando el conjunto FOLLOW, se dice que la gramática es SLR (1)

follow (S) = { $\$$ }, Reducción en  $\$$  y cambio en +, elimina conflictos Gr

Seguir (T) = { $\$.$ , +}, donde  $\$$  es EOF

Reducción de S.) y cambio de ", eliminar conflictos

Sea  $C = \{I_0, I_1, \dots, I_n\}$  ser la colección canónica de elementos LR (0), siendo los estados correspondientes del analizador 0, 1, ..., i, ..., n sin pérdida de generalidad, sea 0 el estado inicial del analizador (que contiene el artículo  $[S' \rightarrow S]$ )

Las acciones de análisis para el estado i se determinan de la siguiente manera

1. Si  $([A \rightarrow a.a_3] \in I_i) \ \&\& \ ([A \rightarrow na_3] \in I_i)$  sea t ACTION  $[i, a] = \text{shift } j$  \* a es un símbolo terminal /

2. Si  $([A \rightarrow n.] \in I_i)$  sea ACTION  $[i, a] = \text{reduce } A_n$ , para todo a e sigue (A)

3. Si  $([S' \rightarrow S.] \in I_i)$  Ajuste ACCIÓN  $[i, \$] = \text{aceptar}$

Los conflictos S-R o R-R en la tabla implican que la gramática no es SLR (1)

4. Si  $([A \rightarrow .A_3] \in I_i) \ \&\& \ ([A \rightarrow A.3] \in I_i)$  sea GOTO  $[i, A] = j$  "A es un símbolo no terminal /

Todas las demás entradas no definidas por las reglas anteriores se cometen errores

El proceso de construcción del analizador SLR (1) no recuerda suficiente contexto izquierdo para resolver conflictos

En la gramática "L = A" (diapositiva anterior), el símbolo '=' se en seguir (R) debido a la siguiente derivación:

$S' \Rightarrow S = L = R = L = L = L = \text{id} = R = \text{id} = \dots$

La producción utilizada es  $L * R$

La siguiente derivación más a la derecha en reversa no existe (y por lo tanto la reducción por R Lon '=' en el estado 2 es ilegal)  $\text{id} = \text{id} L = \text{id} R = \text{id} \dots$

Generalización del ejemplo anterior

En algunas situaciones, cuando aparece un estado i encima del pila, un prefijo viable  $3a$  puede estar en la pila tal que  $3A$  no puede ir seguido de un 'en cualquier forma de oración correcta

Por lo tanto, la reducción de A o sería inválida en 'a' En el ejemplo anterior.  $= .a = L$ . y  $A = R$ : L no puede ser reducido a R en =, ya que conduciría a lo anterior ilegal secuencia de derivación

Los elementos LR (1) tienen la forma  $[A \rightarrow a.B. a]$ , un ser el símbolo de "anticipación"

Los símbolos de anticipación no juegan ningún papel en los elementos de turno, pero en reducir elementos de la forma  $[A \dots$

$A \rightarrow a$  es válido solo si el siguiente símbolo de entrada es 'a'  $\rightarrow Q$ , a], reducción por

Un elemento LR (1)  $[A \rightarrow a.3. a]$  es válido para un prefijo viable  $\gamma$ , si hay una derivación  $S = SAw = rm da 3W$ , donde,  $\gamma = da$ ,  $a =$  primero ( $w$ ) o  $w = e$  y  $a \$$

Considere la gramática:  $S \rightarrow S. S aSb$  e  $ES a.Sb. S]$  es válido para el VP  $a$ ,  $S \rightarrow S = aSb$

$ES a.Sb. b$  es válido para el VP  $aa$ ,

$S \rightarrow S aSb aaSbb$

$ES . \$$  es válido para el VP  $S S = (S aSb. B$  es válido para el VP  $aaSb$ .

$S \rightarrow S = aSb aaSbb$

## VIDEO DE ANÁLISIS SINTÁCTICO 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)

La mayoría de las gramáticas de los lenguajes de programación también son LALR (1), si son LR (1)

La parte central de los elementos LR (1) (la parte después de omitir el símbolo de anticipación) es el mismo para varios estados LR (1) (los símbolos de los serán diferentes)

Fusionar los estados con el mismo núcleo, junto con los símbolos de anticipación y cámbiese el nombre

Las partes ACTION y GOTO de la tabla del analizador serán

Modificado Fusionar las filas de la tabla del analizador correspondientes a estados fusionados, reemplazando los antiguos nombres de estados por los correspondientes nuevos nombres para los estados fusionados

Por ejemplo, si los estados 2 y 4 se fusionan en un nuevo estado 24, y los estados 3 y 6 se fusionan en un nuevo estado 36, todas las referencias a estados 2, 4, 3 y 6 serán reemplazados por 24, 24, 36 y 36, respectivamente

Los analizadores LALR (1) pueden realizar algunas reducciones más (pero no cambia) que un analizador LR (1) antes de detectar un error

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

Los estados del analizador LR (1) y LALR (1) tienen los mismos elementos básicos (los lookaheads pueden no ser los mismos)

Si el estado  $s_1$  del analizador LALR (1) tiene un conflicto S-R, deben tener dos elementos ( $A \rightarrow \alpha \dots a$  y  $[B \rightarrow \beta \dots a]$ )

Uno de los estados  $s_1'$ , a partir del cual se genera  $s_1$ , debe tener los mismos elementos básicos que  $s_1$  el elemento  $[A \rightarrow \alpha \dots a]$  está en  $s_1'$ , entonces  $s_1'$  también debe tener el elemento  $[B \rightarrow \beta \dots a]$  (la búsqueda anticipada no necesita ser  $a$  en  $s_1'$  - puede ser  $a$  en algún otro estado, pero eso no es de interés para algunos).

Estos dos elementos en  $s_1'$  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 un nuevo conflicto R-R 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:

$S \rightarrow S \mid S a A d \mid b B d \mid a B e \mid b A e$

$A \rightarrow c \mid B \mid c$

Dos estados contienen los elementos:

$\{[A \rightarrow c \dots d], [B \rightarrow C m e]\}$  y

$\{[A \rightarrow C \dots e], [B \rightarrow C \dots d]\}$

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

$\{[A \rightarrow C \dots d e], [B \rightarrow C \dots d e]\}$

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  $A a$  donde  $A$  es un no terminal mayor y  $a$  es una cadena adecuada de símbolos gramaticales (generalmente terminal símbolos), posiblemente vacío

Asocia una rutina de mensajes de error con cada error.



Producción Crea un analizador LALR (1) para la nueva gramática con error producciones.

Cuando el analizador encuentra un error, escanea la pila para encontrar el estado superior que contiene un elemento de error del formulario

Un error a

El analizador luego cambia un error de token como si hubiera ocurrido en la entrada Si  $a = \epsilon$ , reduce en  $A \rightarrow e$ , e invoca el mensaje de error rutina asociada con ella. Si es un  $e$ , descarta los símbolos de entrada hasta que encuentra un símbolo con que el analizador puede proceder

La reducción por A. error n ocurre en el momento apropiado

Ejemplo: Si la producción del error es  $A \rightarrow .error;$ , entonces el

El analizador salta los símbolos de entrada hasta; se encuentra, realiza reducción por A  $.error$ , y procede como arriba o La recuperación de errores no es perfecta y el analizador puede abortar al final de entrada.

Tokens:% token nombre1 nombre2 nombre3, ...

Símbolo de inicio:% nombre de inicio

Nombres en reglas: letra (letra | dígito | . | \_) \* la letra es un carácter en minúscula o en mayúscula

- Valores de símbolos y acciones: ejemplo

A: B

{ $$$ = 1$ ; }

C.  $X = \$ 2$ ;  $y = \$ 3$ ;  $$$ = x + y$ ;

- Ahora, el valor de A se almacena en  $$$$  (segundo), el de B en.

$\$ 1$ , el de la acción 1 en  $\$ 2$  y el de C en  $\$ 3$ .

La acción intermedia en el ejemplo anterior se traduce en una producción electrónica de la siguiente manera:

$\$ ACT1: / * vacío * /$

( $$$ = 1$ ;

;

A: B  $\$ ACT1$  C

{ $x = \$ 2$ ;  $y = \$ 3$ ;  $$$ = x + y$ ; }

o Las acciones intermedias pueden devolver valores

Por ejemplo, el primer  $$$$  en el ejemplo anterior es disponible como  $\$ 2$

Sin embargo, las acciones intermedias no pueden referirse a valores de símbolos a la izquierda de la acción. Las acciones se traducen a código C que se ejecutan solo antes de que el analizador realice una reducción

LA devuelve enteros como números token

Los números de token son asignados automáticamente por YACC, a partir de 257, para todos los tokens declarados utilizando % token declaración

Los tokens pueden devolver no solo números de token sino también otra información (por ejemplo, valor de un número, cadena de caracteres de un nombre, puntero a la tabla de símbolos, etc.)

Se devuelven valores adicionales en la variable, yyval, conocida por Analizadores generados por YACC

OE E + EE-EE E E / E | (E) | carné de identidad

- Ambigüedad con asociatividad izquierda o derecha de - 'y /
- Esto causa conflictos de reducción de cambios en YACC: (E-E-E) – cambio o reducir en -?
- Regla de desambiguación en YACC:
- El valor predeterminado es la acción de cambio en los conflictos S-R
- Reducir por regla anterior en conflictos R-R
- La asociatividad se puede especificar explícitamente
- De manera similar, la precedencia de los operadores causa conflictos S-R.

También se puede especificar la precedencia

Los tokens y los no terminales son símbolos de pila. Los símbolos de pila se pueden asociar con valores cuyos tipos se declaran en una declaración de % de unión en el YACC archivo de especificación

- YACC convierte esto en un tipo de unión llamado YYSTYPE
- Con declaraciones de % token y % type, informamos a YACC

Sobre los tipos de valores que toman los tokens y los no terminales. Automáticamente, las referencias a \$ 1, \$ 2, yval, etc., se refieren a el miembro apropiado del sindicato (ver ejemplo a continuación). Para evitar una cascada de mensajes de error, el analizador permanece en estado de error (después de ingresarlo) hasta que los tokens se han transferido con éxito a la pila. En caso de que ocurra un error antes de esto, no hay más mensajes se dan y el símbolo de entrada (que causa el error) es silenciosamente eliminado

El usuario puede identificar los principales no terminales, como los para programa, declaración o bloque, y agregar error producciones para estos a la gramática

Ejemplos

error de declaración (acción 1}

error de declaración; (acción 2}

## VIDEO DE ANÁLISIS SEMÁNTICO CON GRAMÁTICA DE ATRIBUTOS 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**.

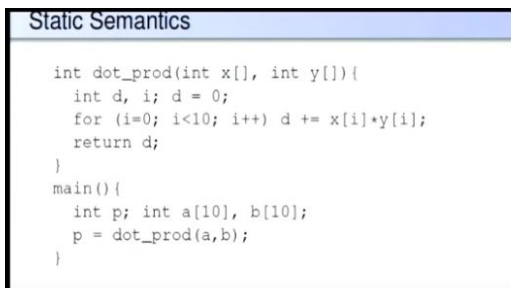
**Semántica estática:** como su nombre indica, usted sabe que no depende del sistema de tiempo de ejecución y la ejecución del programa, sino que depende únicamente de la definición del lenguaje de programación.

**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.

El compilador de semántica dinámica sólo puede generar código para verificar el significado del programa. por ejemplo, si se producirá un desbordamiento durante una operación aritmética, si los límites de la matriz se cruzaran durante la ejecución, si la recursividad cruzará los límites de la pila, si la memoria del montón será suficiente.

Ejemplo:



```
Static Semantics

int dot_prod(int x[], int y[]){
    int d, i; d = 0;
    for (i=0; i<10; i++) d += x[i]*y[i];
    return d;
}

main(){
    int p; int a[10], b[10];
    p = dot_prod(a,b);
}
```

Muestras de comprobaciones semánticas estáticas en main

- Tipos de py return tipos de coincidencia dot\_prod
- El número y tipo de los parámetros de dor\_prod son los mismos tanto en su declaración como en su uso
- p se declara antes de su uso, lo mismo para a y b

## Static Semantics

```
int dot_prod(int x[], int y[]){
    int d, i; d = 0;
    for (i=0; i<10; i++) d += x[i]*y[i];
    return d;
}
main(){
    int p; int a[10], b[10];
    p = dot_prod(a,b);
}
```

Muestras de comprobaciones semánticas estáticas en dot\_prod

- d e i se declaran antes de su uso
- Tipo de d coincide con el tipo de retorno de dot\_prod
- El tipo de d coincide con los tipos de resultado de "\*"
- Los elementos de las matrices x y son compatibles con "\*"

## Dynamic Semantics

```
int dot_prod(int x[], int y[]){
    int d, i; d = 0;
    for (i=0; i<10; i++) d += x[i]*y[i];
    return d;
}
main(){
    int p; int a[10], b[10];
    p = dot_prod(a,b);
}
```

Muestras de comprobaciones semánticas dinámicas en dot\_prod

- El valor de i no excede el rango declarado de matrices x y (tanto inferior como superior)
- no hay desbordamientos durante las operaciones f "\*" y "+" en d += x[i] \* y[i]

## Dynamic Semantics

```
int fact(int n){
    if (n==0) return 1;
    else return (n*fact(n-1));
}
main(){int p; p = fact(10); }
```

Muestras de controles semánticos dinámicos de fact

- La pila del programa no se desborda debido a la recursividad
- No hay desbordamiento debido a "\*" en n + fact(n-1)

La información de tipos se almacena en la tabla de símbolos o en el árbol de sintaxis se puede almacenar en la tabla de símbolos o se puede almacenar en las notas del árbol de sintaxis en sí, generalmente la opción de tabla de símbolos es mejor porque el árbol de sintaxis ocupa mucho más espacio que la tabla de símbolos, generalmente lo almacenamos en una tabla de símbolos y luego coloque un punto al apropiado en la tabla de símbolos en uno de los campos en los nodos del árbol de sintaxis, tipos de variables, parámetros de función, dimensiones de matriz, por supuesto, el nombre de la variable, ect, también se

almacenan en la tabla de símbolos . por lo que estas tablas de símbolos se utilizan no sólo para la validación semántica o el análisis semántico, sino que también se requieren y se utilizan para posteriores

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  $AI$  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

#### Attribute Grammar - Example 1

- The following CFG  
 $S \rightarrow A B C, A \rightarrow aA \mid a, B \rightarrow bB \mid b, C \rightarrow cC \mid c$   
generates:  $L(G) = \{a^m b^n c^p \mid m, n, p \geq 1\}$
- We define an AG (attribute grammar) based on this CFG to generate  $L = \{a^n b^n c^n \mid n \geq 1\}$
- All the non-terminals will have only synthesized attributes
  - $AS(S) = \{equal \uparrow: \{T, F\}\}$
  - $AS(A) = AS(B) = AS(C) = \{count \uparrow: integer\}$

Gráfico de dependencia de atributos

Sea  $T$  un árbol de análisis sintáctico generado por el CFG de un AG,  $G$ .

El gráfico de dependencia de atributos (gráfico de dependencia para abreviar) para  $T$  es el gráfico dirigido,  $DG(T) = (V.E)$ , donde:

$V = \{b \mid b \text{ es una instancia de atributo de algún nodo de árbol}\}$ , y

$E = \{(b, c) \mid b, c \in V, \text{ byc son atributos de símbolos gramaticales en la misma producción } P \text{ de } B \text{ y el valor de } b \text{ se usa para calcular el valor de } c \text{ en la regla de cálculo de atributos asociada con la producción } p\}$

## VIDEO DE ANÁLISIS SEMÁNTICO CON GRAMÁTICA DE ATRIBUTOS PARTE 2

Sea  $G = (N, T, P, S)$  un CFG y sea  $V = N \cup T$ . 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. Una producción  $p$  en  $P$  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:  $110.101 = 6.625$

•  $N + L.R. \quad L \rightarrow BL \mid B. \quad R \rightarrow BR \mid B. \quad B \rightarrow 0 \mid 1$

•  $AS(N) = AS(R) = AS(B) = \{\text{valor} \uparrow: \text{real}\}.$

$AS(L) = \{\text{longitud} \uparrow: \text{entero}, \text{valor} \uparrow: \text{real}\}$

$N \quad L.R \quad \{N.\text{valor} \uparrow = L.\text{valor} \uparrow - R.\text{valor} \uparrow\}$

$B \quad (L.\text{valor} \uparrow = B.\text{value} \uparrow: L.\text{length} \uparrow = 1)$

$+ \quad BL2 \quad \{L1.\text{longitud} \uparrow = L2.\text{longitud} \uparrow + 1:$

$2 \rightarrow L1.\text{valor} \uparrow = B.\text{valor} \uparrow 2 - 2.\text{longitud} \uparrow + L2.\text{valor} \uparrow\}$

$R \quad B \quad (R.\text{value} \uparrow = B.\text{valor} \uparrow / 2)$

$R \quad BR2 \quad (R.\text{valor} \uparrow = (B.\text{valor} \uparrow + R2.\text{valor} \uparrow) / 2)$

$OB0 \quad (B.\text{valor} \uparrow = 0)$

$B \quad 1 \quad \{B.\text{value} \uparrow: -1\}$

Un AG simple para la evaluación de un número real a partir de su representación de cadena de bits Ejemplo:  $110.1010 = 6 + 10/24 = 6 + 10/16 = 6 + 0,625 = 6,625$

•  $N + X.X. \quad X \rightarrow BX \mid B. \quad B \rightarrow 0 \mid 1$

•  $AS(N) = AS(B) = \{\text{valor} \uparrow: \text{real}\}.$

$AS(X) = \{\text{longitud} \uparrow: \text{entero}, \text{valor} \uparrow: \text{real}\}$

EN X1.X2 {N.valor t: = X ,. valor t + X2. valor t / 2 \* length}

XB (X.valor t B.valor t: X.length ↑: = 1}

XBX2 {X.length t- X2 longitud t +1:

X1.value t B.valuet + 2Xlengthm + X2. valor t}

B-0 (B.valor t: = 0}

B 1 (B.valor t: = 1}

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

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

AS (T) = {tipo t: {integer.real}}

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

ODList D | DLista D

OD TL {L.type T.ltype t}

OT int {T.tipo ↑: = entero}

OT float {T.type t: = real}

OLID {ID.type = L.type |}

OL + L2. ID (L2.type - L1.type ID.type = L1.type}

Identificador de ID (ID.name t identifier.namet}

Ejemplo: int a, b, c; flotar x.y

a, b y c están etiquetados con tipo entero

x.y yz están etiquetados con el tipo real

Consideremos primero el CFG para un lenguaje simple

SIE

O E-E + T | T | sea id = E en (E)

V T F | F

OF (E) número | carné de identidad

Este lenguaje permite que las expresiones se aniden dentro expresiones y tienen alcances para los nombres

Deje  $A = 5$  en  $((\text{deje } A = 6 \text{ en } (A * 7)) - A)$  evalúa correctamente a

37, siendo los alcances de las dos instancias de  $A$  diferentes

Requiere una tabla de símbolos con ámbito para su implementación

Una gramática de atributos abstractos para los usos del lenguaje anterior atributos heredados y sintetizados

Se pueden evaluar tanto los atributos heredados como los sintetizados en una pasada (de izquierda a derecha) sobre el árbol de análisis

Los atributos heredados no se pueden evaluar durante el análisis LR

### **VIDEO DE ANÁLISIS SEMÁNTICO CON GRAMÁTICA DE ATRIBUTOS PARTE 3**

Sea  $G = (N, T, P, S)$  un CFG y sea  $V = NUT$ .

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  $p: A \rightarrow X_1 X_2 \dots X_p$ . Sea  $X_i.a$  e  $AI(X_i)$ .

$X_j.a$  solo puede usar o elementos de  $AI(A)$

Elementos de  $AI(A)$  /  $AI(X_k)$  o  $AS(X_k)$ .  $k = 1 \dots i-1$  (es decir, atributos de  $X_1 \dots X_{i-1}$ )

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

• AI (L) = AI (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 {T.type t = integer}

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

ID {ID.type = L.type}

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

Identificador de ID (ID.name t identifier.name t}

Ejemplo: a, b.c: int, x, y: float

a, b y c están etiquetados con tipo entero

x.y yz están etiquetados con el tipo real

T: =E.val H

E E2 + T {E2.symtab = E, .symtab;

E .val 1: = E2.valt + T.val ↑: T.symtab: = E1.symtab}

OET {T.symtab: = E.symtab 1; E.val ↑: = T.val t}

OE, sea  $id = E_3$  en  $(E_3)$

$\{E.val\ t: = E_3.val\ t: E_2.symtab: = E.symtab:$

$E_3.symtab = E.symtab \setminus \{id.name \uparrow E_2.val \uparrow\}$

Nota: cambiando la producción anterior a:

E, devuelve  $(E_3)$  con  $id = E_2$  (con el mismo  
reglas de cálculo) cambia este AG en no-LAG

$T\ T_2\ F\ (T\ val\ t = T_2.val \uparrow F.val\ t:$

$T_2.symtab = T_1.symtab: F.symtab: = T_1.symtab\}$

$T\ F\ T.val\ t: = F\ val\ t. F.symtab: = T.symtab\}$

$F\ (E)\ \{F.val\ t: = E.val \uparrow E.symtab = F.symtab\}$

Número F  $(F.val\ t\ número.val\ t\}$

OF  $id\ (F.val\ t\ F.symtab\ [id.name\ t\}$

Aparte de las reglas de cálculo de atributos, algunos programas segmento que realiza una salida o algún otro lado se agrega cálculo sin efecto al AG

Algunos ejemplos son: operaciones de tabla de símbolos, escritura generada código a un archivo, etc.

Como resultado de estos segmentos de códigos de acción, la evaluación los pedidos pueden estar limitados

Estas restricciones se agregan a la dependencia del atributo grafica como aristas implícitas

Estas acciones se pueden agregar tanto a SAGS como a LAGS (haciéndolos, SATG y LATG resp.)

Nuestra discusión sobre análisis semántico utilizará LATG (1 paso) y SATG

SATG

1.  $Decl \rightarrow DList\ \$$

2.  $DList \rightarrow D \mid DList; re$

3.  $D\ TL\ \{tipo\ de\ parche\ (T.type\ t.\ L.\ lista\ de\ nombres\ t); \}$

4.  $T\ int\ \{T.type\ \uparrow: = integer\}$

5.  $T > float\ (T.type\ \uparrow: = real)\}$

6.  $L \rightarrow id\ \{sp = insert\_symtab\ (id.name\ \uparrow):$

$L.namelist\ t\ makelist\ (sp): \}$

7. L + L2. id {sp = insert\_syntab (id.name t);

L1.namelist t- append (L2.namelist t. Sp):}

Decl DList \$

DList D D; DList

OD TL

OTint | flotador

OID ARR ID ARR. L

ID ID ARR | id | DIMLIST || id BR\_DIMLIST

O DIMLIST → núm núm. DIMLIST

O BR DIMLIST num ||| num BR DIMLIST

Nota: las declaraciones de matriz tienen dos posibilidades

int a [10,20,30]; flotador b [25] [35];

La gramática no es LL (1) y, por tanto, un analizador de LL (1) no se puede construir a partir de él.

Suponemos que el árbol de análisis está disponible y que la evaluación de atributos se realiza sobre el árbol de análisis

Modificaciones al CFG para convertirlo en LL (1) y ellos cambios correspondientes al AG se dejan como ejercicios

Los atributos y sus reglas de cálculo para las producciones 1-4 son como antes y las ignoramos

Proporcionamos el AG solo para las producciones 5-7; AG por regla 8 es similar al de la regla 7

El manejo de declaraciones constantes es similar al manejo declaraciones de variables

1. tipo: (simple, matriz)

2. type = simple para nombres que no son arreglos

3. Los campos eletype y dimlist\_ptr son relevantes solo para matrices. En ese caso, escriba matriz

4. eletype :. (integer, real, errortype), es el tipo de una identificación simple el tipo de elemento de la matriz

5. dimlist\_ptr apunta a una lista de rangos de las dimensiones de una formación. Se asumen declaraciones de matriz de tipo C

Ex. flotar mi matriz [5 || 12 || 15]

dimlist ptr apunta a la lista (5,12.15), y el número total

Elementos en la matriz es 5x12x15 900, que puede ser obtenido recorriendo esta lista y multiplicando los elementos.

Registro de información de tipo de identificador

1. nombre
2. tipo
3. eletype
4. dimlist\_ptr

### 3.2.2 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  $L(G)$ . El conjunto de lenguajes definidos por gramáticas libres de contexto se denomina conjunto de lenguajes libres de contexto

- La primera regla o producción dice "SheepNoise puede derivar la palabra baa seguida de más SheepNoise".

SheepNoise es una variable sintáctica que representa el conjunto de cadenas que se pueden derivar de la gramática. A esta variable sintáctica la llamamos símbolo no terminal. Cada palabra en el idioma definido por la gramática es un símbolo terminal.

- La segunda regla dice "SheepNoise también puede derivar la cadena baa".

### 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.

**S** es un no terminal designado como símbolo de meta o símbolo de inicio de la gramática.  $S$  representa el conjunto de oraciones en  $L(G)$ .

**P** es el conjunto de producciones o reglas de reescritura en **G**. Cada regla en **P** tiene la forma  $NT \rightarrow (T \cup NT)^+$ ; es decir, reemplaza un único no terminal con una cadena de uno o más símbolos gramaticales.

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  $\rightarrow$  (bracket) bracket  $\rightarrow$  [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. Para permitir cualquiera de las dos, tendríamos que introducir un nuevo símbolo Inicio y las producciones Inicio  $\rightarrow$  Paren | Soporte.

## 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 envolviéndolos 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,  $\alpha$ , en la cadena del prototipo, elegimos una regla gramatical,  $\alpha \rightarrow \beta$ , y reescribimos  $\alpha$  con  $\beta$ . Repetimos este 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. En cada punto de este proceso de derivación, la cadena es una colección de símbolos terminales o no terminales. Tal cadena se llama forma enunciada si ocurre en algún paso de una derivación válida. Cualquier forma oracional puede derivarse del símbolo de inicio en cero o más pasos. De manera similar, de cualquier forma de oración podemos derivar una oración válida en cero o más pasos. Por lo tanto, si comenzamos con

SheepNoise y aplica reescrituras sucesivas usando las dos reglas, en cada paso del proceso la cadena es una forma de oración. Cuando llegamos al punto donde la cadena contiene solo símbolos terminales, la cadena es una oración en **L** (SN)

Para derivar una oración en SN, comenzamos con la cadena que consta de un símbolo, SheepNoise.

Podemos reescribir SheepNoise con la regla 1 o la regla 2.

Si reescribimos SheepNoise con la regla 2, la cadena se convierte en baa no tiene más oportunidades para reescribir. La reescritura muestra que baa es una oración válida en L (SN).

La otra opción, reescribir la cadena inicial con la regla 1, conduce a una cadena con dos símbolos: baa SheepNoise. A esta cadena le queda una no terminal; reescribir con la regla 2 conduce a la cadena baa baa, que es una oración en L (SN).

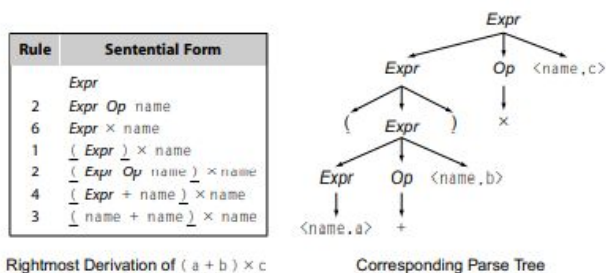
Cómo conveniencia de notación, usaremos  $\rightarrow +$  para significar "deriva en uno o más pasos ". Por lo tanto,  $\text{SheepNoise} \rightarrow + \text{baa}$  y  $\text{SheepNoise} \rightarrow + \text{baa baa}$ .

La regla 1 alarga la cuerda mientras que la regla 2 elimina el SheepNoise no terminal. (La cadena nunca puede contener más de una instancia de SheepNoise).

Todas las cadenas válidas en SN se derivan de cero o más aplicaciones de la regla 1, seguido de la regla 2.

### Ejemplo Complejo:

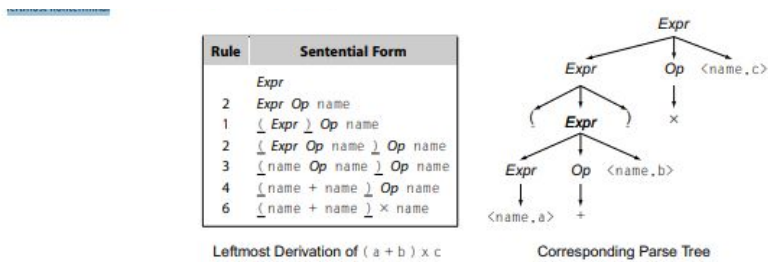
Comenzando con el símbolo de inicio, Expr, podemos generar dos tipos de subterráneos: subterráneos entre paréntesis, con la regla 1, o subterráneos simples, con la regla 2. Para generar la oración "(a + b) × c", podemos usar la siguiente reescritura secuencia (2,6,1,2,4,3), que se muestra a la izquierda. 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  $(a + b) \times c$  una derivación que reescribe, en cada paso, la no terminal más a la izquierda sería:



Las derivaciones más a la izquierda y más a la derecha utilizan el mismo conjunto de reglas; ellos aplican esas reglas en un orden diferente. Porque un árbol de análisis representa las reglas aplicado, pero no el orden de su aplicación, los árboles de análisis para los dos las derivaciones son idénticas.

Si existen múltiples derivaciones más a la derecha (o más a la izquierda) para alguna oración, entonces, en algún punto de la derivación, múltiples reescrituras distintas del no terminal más a la derecha (o más a la izquierda) conducen a la misma oración. Una gramática en la que existen múltiples derivaciones más a la derecha (o más a la izquierda) para una oración se llama gramática ambigua.

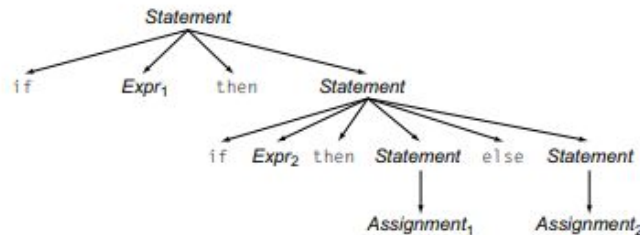
Una gramática ambigua puede producir múltiples derivaciones y múltiples árboles de análisis, dado que las etapas posteriores de la traducción asociaron es decir, con la forma detallada del árbol de análisis sintáctico, varios árboles de análisis sintáctico implican múltiples significados posibles para un solo programa, una propiedad incorrecta para un lenguaje de programación. Si el compilador no puede estar seguro del significado de una oración, no puede traducirla a una secuencia de código definitiva.

El ejemplo clásico de una construcción ambigua en la gramática de un lenguaje de programación es la construcción if-then-else

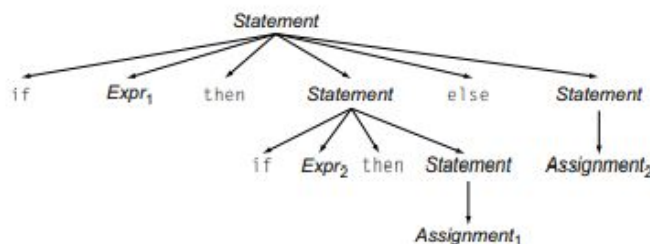
This fragment shows that the `else` is optional. Unfortunately, the code fragment

```
if Expr1 then if Expr2 then Assignment1 else Assignment2
```

has two distinct rightmost derivations. The difference between them is simple. The first derivation has `Assignment2` controlled by the inner `if`, so `Assignment2` executes when `Expr1` is true and `Expr2` is false:



The second derivation associates the `else` clause with the first `if`, so that `Assignment2` executes when `Expr1` is false, independent of the value of `Expr2`:



Clearly, these two derivations produce different behaviors in the compiled code.

## 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.

El sistema de tipos crea un segundo vocabulario para describir tanto la forma como el comportamiento de programas válidos. Analizar un programa desde la perspectiva de su sistema de tipos produce información que no se puede obtener utilizando las técnicas de



escaneo y análisis sintáctico. En un compilador, esta información se usa generalmente para tres propósitos distintos: seguridad, expresividad y eficiencia en tiempo de ejecución.

Type of			Code
a	b	a+b	
integer	integer	integer	iADD $r_a, r_b \Rightarrow r_{a+b}$
integer	real	real	i2f $r_a \Rightarrow r_{af}$ fADD $r_{af}, r_b \Rightarrow r_{af+b}$
integer	double	double	i2d $r_a \Rightarrow r_{ad}$ dADD $r_{ad}, r_b \Rightarrow r_{ad+b}$
real	real	real	fADD $r_a, r_b \Rightarrow r_{a+b}$
real	double	double	r2d $r_a \Rightarrow r_{ad}$ dADD $r_{ad}, r_b \Rightarrow r_{ad+b}$
double	double	double	dADD $r_a, r_b \Rightarrow r_{a+b}$

■ FIGURE 4.2 Implementing Addition in FORTRAN 77.

## Type Checking

Para evitar la sobrecarga de la verificación de tipos en tiempo de ejecución, el compilador debe analizar el programa y asignar un tipo a cada nombre y expresión. Debe comprobar estos tipos para asegurarse de que se utilizan en contextos donde son legales. En conjunto, estas actividades a menudo se denominan verificación de tipos. Este es un nombre poco afortunado, porque agrupa las actividades separadas de la inferencia de tipos y la identificación de errores relacionados con los tipos.

Un lenguaje fuertemente tipado y comprobable estáticamente puede implementarse con verificación en tiempo de ejecución (o sin verificación). Un lenguaje sin tipear puede implementarse de manera que detecte ciertos tipos de errores. Tanto ml como Modula-3 son buenos ejemplos de lenguajes fuertemente tipados que se pueden verificar estáticamente. Common Lisp tiene un sistema de tipos fuerte que debe comprobarse dinámicamente. ansi c es un lenguaje escrito, pero algunas implementaciones no identifican errores de tipo.

La teoría que subyace a los sistemas de tipos abarca un cuerpo de conocimiento amplio y complejo.

### 4.2.2 Componentes de un sistema de tipos

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 contruidos

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  $100 \times 200 = 20.000$  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  $10 \times 10$  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  $10 \times 10$  tiene un tipo diferente que una matriz de enteros de  $12 \times 12$ . 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 contruidos.

- 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.

- Punteros

Los punteros son direcciones de memoria abstractas que permiten al programador manipular estructuras de datos arbitrarias. Muchos idiomas incluyen un tipo de puntero. Los punteros permiten que un programa guarde una dirección y luego examine el objeto al que se dirige.

Los punteros se crean cuando se crean objetos (nuevos en Java o malloc en c).

Para proteger a los programadores de usar un puntero para escribir t para hacer referencia a una estructura de tipo s, algunos lenguajes restringen la asignación de punteros a tipos "equivalentes".

En estos idiomas, el puntero del lado izquierdo de una asignación debe tener el mismo tipo que la expresión del lado derecho. Un programa puede asignar legalmente un puntero a entero a una variable declarada como puntero a entero pero no a una declarada como puntero a puntero a entero o puntero a booleano.

- Equivalencia de tipo

Un componente crítico de cualquier sistema de tipos es el mecanismo que utiliza para decidir si dos declaraciones de tipos diferentes son equivalentes o no. Considere las dos declaraciones en c que se muestran al margen. ¿Tree y STree son del mismo tipo? ¿Son equivalentes? Cualquier lenguaje de programación con un sistema de tipos no triviales debe incluir una regla inequívoca para responder a esta pregunta para tipos arbitrarios.

- Reglas de inferencia

En general, las reglas de inferencia de tipos especifican, para cada operador, el mapeo entre los tipos de operandos y el tipo de resultado. En algunos casos, el mapeo es simple. Una asignación, por ejemplo, tiene un operando y un resultado. El resultado, o el lado izquierdo, debe tener un tipo que sea compatible con el tipo de operando, o el lado derecho. (En Pascal, el subrango 1..100 es compatible con los enteros ya que cualquier elemento del subrango puede asignarse de forma segura a un entero). Esta regla permite la asignación de un valor entero a una variable entera. Prohíbe la asignación de una estructura a una variable entera, sin una conversión explícita que dé sentido a la operación.

- Declaraciones e inferencias

Como se mencionó anteriormente, muchos lenguajes de programación incluyen una regla de "declarar antes de usar". Con declaraciones obligatorias, cada variable tiene un tipo bien definido. El compilador necesita una forma de asignar tipos a constantes. Dos enfoques son comunes. O la forma de una constante implica

un tipo específico, por ejemplo, 2 es un número entero y 2.0 es un número de punto flotante, o el compilador infiere un

el tipo de constante de su uso; por ejemplo, sin (2) implica que 2 es un número de punto flotante, mientras que  $x \leftarrow 2$ , para el número entero  $x$ , implica que 2 es un número entero.

Con tipos declarados para variables, tipos implícitos para constantes y un conjunto completo de reglas de inferencia de tipo, el compilador puede asignar tipos a cualquier expresión sobre variables y constantes. Las llamadas a funciones complican el panorama, como veremos.

- Inferir tipos de expresiones

El objetivo de la inferencia de tipos es asignar un tipo a cada expresión que ocurre en un programa. El caso más simple para

La inferencia de tipo ocurre cuando el compilador

puede asignar un tipo a cada elemento base en una expresión

es decir, a cada hoja del árbol de análisis sintáctico para una expresión. Esto requiere declaraciones para todas las variables, tipos inferidos para todas las constantes e información de tipo sobre todas las funciones.

La inferencia de tipos para expresiones, en este caso simple, sigue directamente la estructura de la expresión. Las reglas de inferencia describen el problema en términos del idioma de origen. La estrategia de evaluación opera de abajo hacia arriba en el árbol de análisis. Por estas razones, la inferencia de tipos para expresiones se ha convertido en un problema de ejemplo clásico para ilustrar el análisis sensible al contexto.

- Aspectos interprocedimientos de la inferencia de tipos

La inferencia de tipos para expresiones depende, inherentemente, de los otros procedimientos que forman el programa ejecutable. Incluso en los sistemas de tipos más simples, las expresiones contienen llamadas a funciones. El compilador debe verificar cada una de esas llamadas. Debe asegurarse de que cada parámetro real sea de tipo compatible con el parámetro formal correspondiente. Debe determinar el tipo de cualquier valor devuelto para su uso en inferencias posteriores.

Para analizar y comprender las llamadas a procedimientos, el compilador necesita una firma de tipo para cada función. Por ejemplo, la función `strlen` en la biblioteca estándar de `c` toma un operando de tipo `char *` y devuelve un `int` que contiene su longitud en bytes, excluyendo el carácter de terminación. En `c`, el prototipo de función del programador puede registrar este hecho con un prototipo de función que se parece a:

```
unsigned int strlen(const char *s);
```

Este prototipo afirma que `strlen` toma un argumento de tipo `char *`, que no modifica, como lo indica el atributo `const`. La función devuelve un número entero no negativo. Escribiendo esto en una notación más abstracta, podríamos decir que:

`strlen: const char * → unsigned int`

que leemos como "strlen es una función que toma una cadena de caracteres de valor constante y devuelve un entero sin signo". Como segundo ejemplo, el clásico

El filtro de función de esquema tiene la firma de tipo

filtro:  $(\alpha \rightarrow \text{booleano}) \times \text{lista de } \alpha \rightarrow \text{lista de } \alpha$

`filter` es una función que toma dos argumentos. La primera debería ser una función que mapee algún tipo  $\alpha$  en un booleano, escrito  $(\alpha \rightarrow \text{booleano})$ , y la segunda debería ser una lista cuyos elementos sean del mismo tipo  $\alpha$ . Dados argumentos de esos tipos, `filter` devuelve una lista cuyos elementos tienen el tipo  $\alpha$ .

El filtro de función exhibe polimorfismo paramétrico: su tipo de resultado es una función de sus tipos de argumentos.

## 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 código LOC 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.

El ir puede variar desde una representación de fuente cercana en la que un solo nodo podría representar un acceso a una matriz o una llamada a un procedimiento hasta una representación de bajo nivel en la que varias operaciones de ir deben combinarse para formar una sola operación de máquina objetivo.

El escritor del compilador debe considerar la expresividad del ir, su capacidad para acomodar todos los hechos que el compilador necesita registrar. El ir de un procedimiento puede incluir el código que lo define, los resultados del análisis estático, datos de perfil de ejecuciones anteriores y mapas para permitir que el depurador comprenda el código y sus datos. Todos estos hechos deben expresarse de una manera que deje clara su relación con puntos específicos en el IR.

### 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.

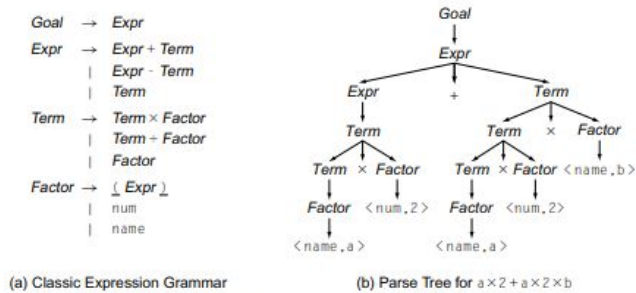
#### Árboles relacionados con la sintaxis

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  $a \times 2 + a \times 2 \times b$ . 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  $a \times 2 + a \times 2 \times b$ :



ASTs se ha utilizado en muchos sistemas de compilación prácticos. Los sistemas de fuente a fuente, incluidos los editores dirigidos por sintaxis y las herramientas de paralelización automática, a menudo utilizan un ast desde el que se puede regenerar fácilmente el código fuente. Las expresiones S que se encuentran en las implementaciones de Lisp y Scheme son, esencialmente, asts.

Si el compilador genera código fuente como salida, el ast suele tener abstracciones a nivel de fuente. Si el compilador genera código ensamblador, la versión final del ast suele estar en el nivel de abstracción del conjunto de instrucciones de la máquina o por debajo de él.

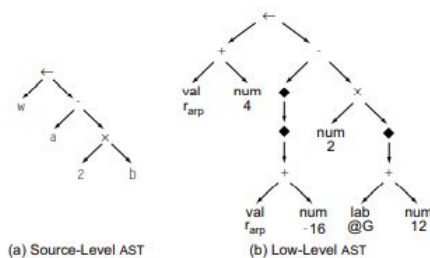
## Gráficos acíclicos dirigidos

Si bien el ast es más conciso que un árbol de sintaxis, conserva fielmente la estructura del código fuente original. Por ejemplo, el ast para  $a \times 2 + a \times 2 \times b$  contiene dos copias distintas de la expresión  $a \times 2$ . Un gráfico acíclico dirigido (dag) es una contracción del ast que evita esta duplicación. En un dag, los nodos pueden tener múltiples padres y se reutilizan subárboles idénticos. Tal compartir hace que el dag sea más compacto que el correspondiente ast.

Los dags se utilizan en sistemas reales por dos razones. Si las restricciones de memoria limitan el tamaño de los programas que puede manejar el compilador, el uso de un dag puede ayudar a reducir la huella de memoria. Otros sistemas usan dags para exponer redundancias. Aquí, el beneficio radica en un código mejor compilado. Estos últimos sistemas tienden a usar el dag como un ir derivado, construyendo el dag, transformando el ir definitivo para reflejar las redundancias y descartando el dag.

## 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  $w \leftarrow a - 2 \times b$ . 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 num 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. Un gráfico de dependencia de datos encarna esta relación. Los nodos en un gráfico de dependencia de datos representan operaciones. La mayoría de las operaciones contienen tanto definiciones como usos. Un borde en un gráfico de dependencia de datos conecta dos nodos, uno que define un valor y otro que lo usa. Dibujamos gráficos de dependencia con aristas que van desde la definición hasta el 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.

Creación de un gráfico de flujo de control de un código lineal

❖ Como primer paso, el compilador debe encontrar el principio y el final de cada bloque básico en el ir lineal. Llamaremos líder a la operación inicial de un bloque. Una operación es líder si es la primera operación en el procedimiento, o si tiene una etiqueta que es, potencialmente, el objetivo de alguna rama. El compilador puede identificar líderes en una sola pasada sobre el ir. Itera sobre las operaciones en el programa, en orden, encuentra las declaraciones etiquetadas y las registra como líderes. Si el lineal ir contiene etiquetas que no se utilizan como destinos de bifurcación, entonces tratar las etiquetas como líderes puede dividir bloques innecesariamente. El algoritmo podría rastrear qué etiquetas son objetivos de salto. Sin embargo, si el código contiene saltos ambiguos, debe tratar todas las declaraciones etiquetadas como líderes de todos modos.

❖ La segunda pasada, encuentra cada operación de final de bloque. Se asume que cada bloque termina con una rama o un salto y que ramas tomadas "etiqueta". Esto simplifica el manejo de bloques y permite que el compilador back-end para elegir qué ruta será el caso de "caída" de una rama.

(Por el momento, suponga que las ramas no tienen ranuras de retraso).

Para encontrar el final de cada bloque, el algoritmo itera a través de los bloques, en el orden de aparición en la matriz Leader. Camina hacia adelante a través del ir hasta que encuentra al líder del siguiente bloque. La operación inmediatamente anterior a ese líder finaliza el bloque actual. El algoritmo registra el índice de esa operación en Last [i], de modo que el par hLeader [i], Last [i] describe el bloque i.

Agrega bordes al cfg según sea necesario.

## Modelos de memoria

Así como el mecanismo para nombrar valores temporales afecta la información que se puede representar en una versión ir de un programa, también lo hace la elección del compilador de una ubicación de almacenamiento para cada valor.

El compilador debe determinar, para cada valor calculado en el código, dónde residirá ese valor. Para que se ejecute el código, el compilador debe asignar una ubicación específica, como el registro r13 o 16 bytes de la etiqueta L0089. Sin embargo, antes de las etapas finales de la generación de código, el compilador puede usar direcciones simbólicas que codifican un nivel en la jerarquía de memoria, por ejemplo, registros o memoria, pero no una ubicación específica dentro de ese nivel.

En general, los compiladores funcionan a partir de uno de dos modelos de memoria.

### 1. Modelo de registro a registro

En este modelo, el compilador mantiene los valores en los registros de forma agresiva, ignorando las limitaciones impuestas por el tamaño del conjunto de registros físicos de la máquina. Cualquier valor que pueda mantenerse legalmente en un registro durante la mayor parte de su vida se mantiene en un registro. Los valores se almacenan en la memoria sólo cuando la semántica del programa lo requiere; por ejemplo, en una llamada a un procedimiento, cualquier variable local cuya dirección se pasa como parámetro al procedimiento llamado debe volver a almacenarse en la memoria. Un valor que no se puede mantener en un registro durante la mayor parte de su vida se almacena en la memoria. El compilador genera código para almacenar su valor cada vez que se calcula y para cargar su valor en cada uso.

### 2. Modelo de memoria a memoria

En este modelo, el compilador supone que todos los valores se guardan en ubicaciones de memoria. Los valores se mueven de la memoria a un registro justo antes de ser utilizados. Los valores se mueven de un registro a la memoria justo después de su definición. El número de registros nombrados en la versión ir del código puede ser pequeño en comparación con el modelo de registro a registro. En este modelo, el diseñador puede encontrar que vale la pena incluir operaciones de memoria a memoria, como una adición de memoria a memoria, en IR

## Tabla de símbolo

Este repositorio central, llamado tabla de símbolos, se convierte en una parte integral del ir del compilador. La tabla de símbolos localiza la información derivada de partes potencialmente distantes del código fuente. Hace que dicha información esté disponible de manera fácil y eficiente, y simplifica el diseño y la implementación de cualquier código que deba referirse a información sobre variables derivadas anteriormente en la compilación. Evita el gasto de buscar el ir para encontrar la parte que representa la declaración de una variable; El uso de una tabla de símbolos a menudo elimina la necesidad de representar las declaraciones directamente en el ir. (Se produce una excepción en la traducción de fuente a fuente. El compilador puede construir una tabla de símbolos para mayor eficiencia y preservar la sintaxis de declaración en el ir para que pueda producir un programa de salida

que se parezca mucho al programa de entrada). Elimina la sobrecarga haciendo que cada referencia contiene un puntero a la declaración. Reemplaza ambos con un mapeo calculado del nombre textual a la información almacenada. Por lo tanto, en cierto sentido, la tabla de símbolos es simplemente un truco de eficiencia.

La implementación de la tabla de símbolos requiere atención al detalle. Porque casi todos los aspectos de la traducción se refieren a la tabla de símbolos, la eficiencia del acceso es crítico. Debido a que el compilador no puede predecir, antes de la traducción, el número de nombres que encontrará, la expansión de la tabla de símbolos debe ser elegante y eficiente. Esta sección proporciona un tratamiento de alto nivel de los problemas que surgen al diseñar una tabla de símbolos. Presenta los aspectos específicos del compilador del diseño y uso de tablas de símbolos.

## **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. Finalmente, el proceso es la unidad de trabajo básica para la mayoría de los compiladores. Un compilador típico procesa una colección de procesos y genera código para ella, que se vinculará con otras colecciones de procesos compilados y se ejecutará correctamente.

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 se centra en cómo el compilador implementa la abstracción de procesos. Hoja de ruta conceptual Para convertir un programa de lenguaje fuente en código ejecutable, el compilador debe mapear todas las estructuras del lenguaje fuente utilizadas por el programa con las operaciones y estructuras de datos en el procesador de destino. El compilador necesita una estrategia abstracta para el soporte del lenguaje fuente.

Estos

Estas estrategias incluyen algoritmos y estructuras de datos integrados en código ejecutable. Estos algoritmos de tiempo de ejecución y estructuras de datos se combinan para lograr el comportamiento indicado por la abstracción. Estas estrategias de tiempo de ejecución también deben proporcionar soporte en tiempo de compilación en forma de algoritmos y datos. La estructura que se ejecuta dentro 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

Valores o variables pasadas como parámetros en la llamada. El sitio es el parámetro real de la llamada.

#### Parámetros formales

Declarar los nombres de ciertos parámetros producirá un parámetro formado. Variables y etiquetas sin preocuparse por el contexto circundante. En el proceso, estos reclamos locales tienen prioridad sobre cualquier reclamo del mismo nombre. El programador puede crear parámetros para el procedimiento para permitir que la persona que llama mapee valores y variables en el espacio de nombres de la persona que llama a parámetros formales en el espacio de nombres de la persona que llama.

Dado que el procedimiento tiene un espacio de nombres independiente conocido, puede ejecutarse de manera correcta y coherente cuando se llama desde diferentes contextos.

Cuando se realiza la llamada, se crea una instancia del espacio de nombres de la persona que llama. La llamada debe crear almacenamiento para el objeto declarado por el receptor. Esta asignación debe ser automática y eficiente, que es el resultado de llamar al procedimiento.

3. El programa de interfaz externa define la interfaz clave entre las diversas partes del gran sistema de software. Las convenciones de vinculación definen reglas que asignan nombres a valores y ubicaciones, preservan el nombre del entorno de ejecución de la persona que llama y crean el entorno del receptor, y transfieren el control de la persona que llama a la persona que llama y viceversa. Cree un contexto en el que los programadores puedan llamar con seguridad el código escrito por otros.

La existencia de una secuencia de llamadas unificada permite el desarrollo y uso de bibliotecas y llamadas al sistema. Si no hay un acuerdo vinculante,

El programador y el compilador necesitarán comprender la implementación del receptor en detalle en cada llamada a procedimiento.

Por lo tanto, en muchos sentidos, el proceso es similar a la abstracción básica del lenguaje de Algol. Es una apariencia exquisita creada por el compilador y el hardware subyacente con la ayuda del sistema operativo. El proceso crea variables con nombre y las asigna a direcciones virtuales; el sistema operativo asigna direcciones virtuales a direcciones físicas.

Este programa establece reglas para la visibilidad y direccionalidad del nombre; el hardware generalmente proporciona varias variantes de operaciones de carga y almacenamiento. El programa nos permite descomponer grandes sistemas de software en Componentes, el enlazador y el cargador los unen para convertirse en un programa ejecutable. El hardware puede ejecutar aumentando la velocidad de ejecución de su contador de programa y las ramas posteriores.

Una gran parte del trabajo del compilador es implementar el código necesario para descubrir todos los aspectos de la abstracción de procesos. El compilador debe especificar el diseño de la memoria y codificar el diseño en el programa generado. Dado que puede compilar diferentes componentes del programa en diferentes momentos sin comprender la relación entre ellos,

El diseño de la memoria y todas las convenciones que provoca deben estar estandarizados. Y aplicar uniformemente. El compilador también debe usar varias interfaces proporcionadas por el sistema operativo para procesar la entrada y la salida, administrar la memoria y comunicarse con otros procesos.