Capítulo 4

Análisis sintáctico

Este capítulo está dedicado a los métodos de análisis sintáctico que se utilizan, por lo general, en los compiladores. Primero presentaremos los conceptos básicos, después las técnicas adecuadas para la implementación manual y, por último, los algoritmos que se han utilizado en las herramientas automatizadas. Debido a que los programas pueden contener errores sintácticos, hablaremos sobre las extensiones de los métodos de análisis sintáctico para recuperarse de los errores comunes.

Por diseño, cada lenguaje de programación tiene reglas precisas, las cuales prescriben la estructura sintáctica de los programas bien formados. Por ejemplo, en C un programa está compuesto de funciones, una función de declaraciones e instrucciones, una instrucción de expresiones, y así sucesivamente. La sintaxis de las construcciones de un lenguaje de programación puede especificarse mediante gramáticas libres de contexto o la notación BNF (Forma de Backus-Naur), que se presentó en la sección 2.2. Las gramáticas ofrecen beneficios considerables, tanto para los diseñadores de lenguajes como para los escritores de compiladores.

- Una gramática proporciona una especificación sintáctica precisa, pero fácil de entender, de un lenguaje de programación.
- A partir de ciertas clases de gramáticas, podemos construir de manera automática un analizador sintáctico eficiente que determine la estructura sintáctica de un programa fuente. Como beneficio colateral, el proceso de construcción del analizador sintáctico puede revelar ambigüedades sintácticas y puntos problemáticos que podrían haberse pasado por alto durante la fase inicial del diseño del lenguaje.
- La estructura impartida a un lenguaje mediante una gramática diseñada en forma apropiada es útil para traducir los programas fuente en código objeto correcto, y para detectar errores.
- Una gramática permite que un lenguaje evolucione o se desarrolle en forma iterativa, agregando nuevas construcciones para realizar nuevas tareas. Estas nuevas construcciones pueden integrarse con más facilidad en una implementación que siga la estructura gramátical del lenguaje.

4.1 Introducción

En esta sección, examinaremos la forma en que el analizador sintáctico se acomoda en un compilador ordinario. Después analizaremos las gramáticas comunes para las expresiones aritméticas. Las gramáticas para las expresiones son suficientes para ilustrar la esencia del análisis sintáctico, ya que dichas técnicas de análisis para las expresiones se transfieren a la mayoría de las construcciones de programación. Esta sección termina con una explicación sobre el manejo de errores, ya que el analizador sintáctico debe responder de manera adecuada para descubrir que su entrada no puede ser generada por su gramática.

4.1.1 La función del analizador sintáctico

En nuestro modelo de compilador, el analizador sintáctico obtiene una cadena de tokens del analizador léxico, como se muestra en la figura 4.1, y verifica que la cadena de nombres de los tokens pueda generarse mediante la gramática para el lenguaje fuente. Esperamos que el analizador sintáctico reporte cualquier error sintáctico en forma inteligible y que se recupere de los errores que ocurren con frecuencia para seguir procesando el resto del programa. De manera conceptual, para los programas bien formados, el analizador sintáctico construye un árbol de análisis sintáctico y lo pasa al resto del compilador para que lo siga procesando. De hecho, el árbol de análisis sintáctico no necesita construirse en forma explícita, ya que las acciones de comprobación y traducción pueden intercalarse con el análisis sintáctico, como veremos más adelante. Por ende, el analizador sintáctico y el resto de la interfaz de usuario podrían implementarse sin problemas mediante un solo módulo.

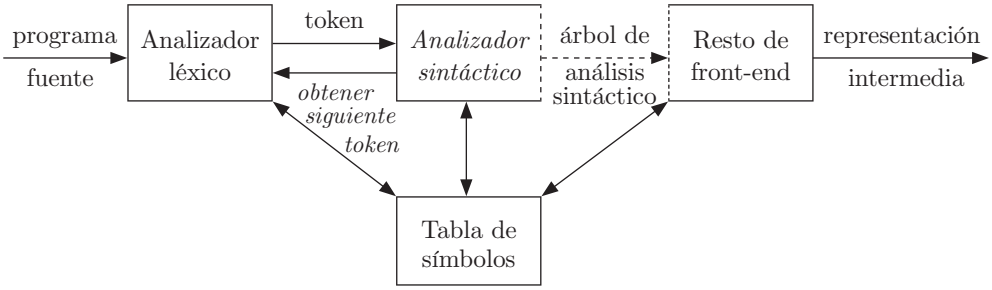

Figura 4.1: Posición del analizador sintáctico en el modelo del compilador

Existen tres tipos generales de analizadores para las gramáticas: universales, descendentes y ascendentes. Los métodos universales de análisis sintáctico como el algoritmo de Cocke-Younger-Kasami y el algoritmo de Earley pueden analizar cualquier gramática (vea las notas bibliográficas). Sin embargo, estos métodos generales son demasiado ineficientes como para usarse en la producción de compiladores.

Los métodos que se utilizan, por lo regular, en los compiladores pueden clasificarse como descendentes o ascendentes. Según sus nombres, los métodos descendentes construyen árboles de análisis sintáctico de la parte superior (raíz) a la parte inferior (hojas), mientras que los métodos ascendentes empiezan de las hojas y avanzan hasta la raíz. En cualquier caso, la entrada al analizador se explora de izquierda a derecha, un símbolo a la vez.

Los métodos descendentes y ascendentes más eficientes sólo funcionan para subclases de gramáticas, pero varias de estas clases, en especial las gramáticas LL y LR, son lo bastante expresivas como para describir la mayoría de las construcciones sintácticas en los lenguajes de programación modernos. Los analizadores sintácticos que se implementan en forma manual utilizan con frecuencia gramáticas LL; por ejemplo, el método de análisis sintáctico predictivo de la sección 2.4.2 funciona para las gramáticas LL. Los analizadores para la clase más extensa de gramáticas LR, por lo general, se construyen mediante herramientas automatizadas.

En este capítulo vamos a suponer que la salida del analizador sintáctico es cierta representación del árbol de análisis sintáctico para el flujo de tokens que proviene del analizador léxico. En la práctica, hay una variedad de tareas que podrían realizarse durante el análisis sintáctico, como la recolección de información de varios tokens para colocarla en la tabla de símbolos, la realización de la comprobación de tipos y otros tipos de análisis semántico, así como la generación de código intermedio. Hemos agrupado todas estas actividades en el cuadro con el título "Resto del front-end" de la figura 4.1. En los siguientes capítulos cubriremos con detalle estas actividades.

4.1.2 Representación de gramáticas

Algunas de las gramáticas que examinaremos en este capítulo se presentan para facilitar la representación de gramáticas. Las construcciones que empiezan con palabras clave como **while** o **int** son muy fáciles de analizar, ya que la palabra clave guía la elección de la producción gramátical que debe aplicarse para hacer que coincida con la entrada. Por lo tanto, nos concentraremos en las expresiones, que representan un reto debido a la asociatividad y la precedencia de operadores.

La asociatividad y la precedencia se resuelvan en la siguiente gramática, que es similar a las que utilizamos en el capítulo 2 para describir expresiones, términos y factores. E representa a las expresiones que consisten en términos separados por los signos +, T representa a los términos que consisten en factores separados por los signos *, y F representa a los factores que pueden ser expresiones entre paréntesis o identificadores:

$$\begin{array}{lcl} E & \rightarrow & E + T \mid T \\ T & \rightarrow & T * F \mid F \\ F & \rightarrow & (E) \mid \mathbf{id} \end{array} \tag{4.1}$$

La gramática para expresiones (4.1) pertenece a la clase de gramáticas LR que son adecuadas para el análisis sintáctico ascendentes. Esta gramática puede adaptarse para manejar operadores adicionales y niveles adicionales de precedencia. Sin embargo, no puede usarse para el análisis sintáctico descendente, ya que es recursiva por la izquierda.

La siguiente variante no recursiva por la izquierda de la gramática de expresiones (4.1) se utilizará para el análisis sintáctico descendente:

$$\begin{array}{lcl} E & \rightarrow & T\ E' \\ E' & \rightarrow & +\ T\ E' \mid \epsilon \\ T & \rightarrow & F\ T' \\ T' & \rightarrow & *\ F\ T' \mid \epsilon \\ F & \rightarrow & (\ E\ ) \mid \mathbf{id} \end{array} \tag{4.2}$$

La siguiente gramática trata a los signos + y ∗ de igual forma, de manera que sirve para ilustrar las técnicas para el manejo de ambigüedades durante el análisis sintáctico:

$$E \rightarrow E + E \mid E * E \mid (E) \mid \mathbf{id} \tag{4.3}$$

Aquí, E representa a las expresiones de todo tipo. La gramática (4.3) permite más de un árbol de análisis sintáctico para las expresiones como $a + b * c$.

4.1.3 Manejo de los errores sintácticos

El resto de esta sección considera la naturaleza de los errores sintácticos y las estrategias generales para recuperarse de ellos. Dos de estas estrategias, conocidas como recuperaciones en modo de pánico y a nivel de frase, se describirán con más detalle junto con los métodos específicos de análisis sintáctico.

Si un compilador tuviera que procesar sólo programas correctos, su diseño e implementación se simplificaría en forma considerable. No obstante, se espera que un compilador ayude al programador a localizar y rastrear los errores que, de manera inevitable, se infiltran en los programas, a pesar de los mejores esfuerzos del programador. Aunque parezca increíble, son pocos los lenguajes que se diseñan teniendo en mente el manejo de errores, aun cuando éstos son tan comunes. Nuestra civilización sería radicalmente distinta si los lenguajes hablados tuvieran los mismos requerimientos en cuanto a precisión sintáctica que los lenguajes de computadora. La mayoría de las especificaciones de los lenguajes de programación no describen la forma en que un compilador debe responder a los errores; el manejo de los mismos es responsabilidad del diseñador del compilador. La planeación del manejo de los errores desde el principio puede simplificar la estructura de un compilador y mejorar su capacidad para manejar los errores.

Los errores de programación comunes pueden ocurrir en muchos niveles distintos.

- Los errores *léxicos* incluyen la escritura incorrecta de los identificadores, las palabras clave o los operadores; por ejemplo, el uso de un identificador `tamanioElipce` en vez de `tamanioElipse`, y la omisión de comillas alrededor del texto que se debe interpretar como una cadena.

- Los errores *sintácticos* incluyen la colocación incorrecta de los signos de punto y coma, además de llaves adicionales o faltantes; es decir, "`{`" o "`}`". Como otro ejemplo, en C o Java, la aparición de una instrucción `case` sin una instrucción `switch` que la encierre es un error sintáctico (sin embargo, por lo general, esta situación la acepta el analizador sintáctico y se atrapa más adelante en el procesamiento, cuando el compilador intenta generar código).

- Los errores *semánticos* incluyen los conflictos de tipos entre los operadores y los operandos. Un ejemplo es una instrucción `return` en un método de Java, con el tipo de resultado `void`.

- Los errores *lógicos* pueden ser cualquier cosa, desde un razonamiento incorrecto del programador en el uso (en un programa en C) del operador de asignación `=`, en vez del operador de comparación `==`. El programa que contenga `=` puede estar bien formado; sin embargo, tal vez no refleje la intención del programador.

La precisión de los métodos de análisis sintáctico permite detectar los errores sintácticos con mucha eficiencia. Varios métodos de análisis sintáctico, como los métodos LL y LR, detectan

un error lo más pronto posible; es decir, cuando el flujo de tokens que proviene del analizador léxico no puede seguirse analizando de acuerdo con la gramática para el lenguaje. Dicho en forma más precisa, tienen la *propiedad de prefijo viable*, lo cual significa que detectan la ocurrencia de un error tan pronto como ven un prefijo de la entrada que no puede completarse para formar una cadena válida en el lenguaje.

Otra de las razones para enfatizar la recuperación de los errores durante el análisis sintáctico es que muchos errores parecen ser sintácticos, sea cual fuere su causa, y se exponen cuando el análisis sintáctico no puede continuar. Algunos errores semánticos, como los conflictos entre los tipos, también pueden detectarse con eficiencia; sin embargo, la detección precisa de los errores semánticos y lógicos en tiempo de compilación es, por lo general, una tarea difícil.

El mango de errores en un analizador sintáctico tiene objetivos que son simples de declarar, pero difíciles de llevar a cabo:

- Reportar la presencia de errores con claridad y precisión.
- Recuperarse de cada error lo bastante rápido como para poder detectar los errores siguientes.
- Agregar una sobrecarga mínima al procesamiento de los programas correctos.

Por fortuna, los errores comunes son simples, y a menudo basta con un mecanismo simple para su manejo.

¿De qué manera un mango de errores debe reportar la presencia de un error? Como mínimo, debe reportar el lugar en el programa fuente en donde se detectó un error, ya que hay una buena probabilidad de que éste en sí haya ocurrido en uno de los pocos tokens anteriores. Una estrategia común es imprimir la línea del problema con un apuntador a la posición en la que se detectó el error.

4.1.4 Estrategias para recuperarse de los errores

Una vez que se detecta un error, ¿cómo debe recuperarse el analizador sintáctico? Aunque no hay una estrategia que haya demostrado ser aceptable en forma universal, algunos métodos pueden aplicarse en muchas situaciones. El método más simple es que el analizador sintáctico termine con un mensaje de error informativo cuando detecte el primer error. A menudo se descubren errores adicionales si el analizador sintáctico puede restaurarse a sí mismo, a un estado en el que pueda continuar el procesamiento de la entrada, con esperanzas razonables de que un mayor procesamiento proporcione información útil para el diagnóstico. Si los errores se apilan, es mejor para el compilador desistir después de exceder cierto límite de errores, que producir una molesta avalancha de errores "falsos".

El resto de esta sección se dedica a las siguientes estrategias de recuperación de los errores: modo de pánico, nivel de frase, producciones de errores y corrección global.

Recuperación en modo de pánico

Con este método, al describir un error el analizador sintáctico descarta los símbolos de entrada, uno a la vez, hasta encontrar un conjunto designado de *tokens de sincronización*. Por lo general, los tokens de sincronización son delimitadores como el punto y coma o `}`, cuya función en

el programa fuente es clara y sin ambigüedades. El diseñador del compilador debe seleccionar los tokens de sincronización apropiados para el lenguaje fuente. Aunque la corrección en modo de pánico a menudo omite una cantidad considerable de entrada sin verificar errores adicionales, tiene la ventaja de ser simple y, a diferencia de ciertos métodos que consideraremos más adelante, se garantiza que no entrará en un ciclo infinito.

Recuperación a nivel de frase

Al descubrir un error, un analizador sintáctico puede realizar una corrección local sobre la entrada restante; es decir, puede sustituir un prefijo de la entrada restante por alguna cadena que le permita continuar. Una corrección local común es sustituir una coma por un punto y coma, eliminar un punto y coma extraño o insertar un punto y coma faltante. La elección de la corrección local se deja al diseñador del compilador. Desde luego que debemos tener cuidado de elegir sustituciones que no nos lleven hacia ciclos infinitos, como sería, por ejemplo, si siempre insertáramos algo en la entrada adelante del símbolo de entrada actual.

La sustitución a nivel de frase se ha utilizado en varios compiladores que reparan los errores, ya que puede corregir cualquier cadena de entrada. Su desventaja principal es la dificultad que tiene para arreglárselas con situaciones en las que el error actual ocurre antes del punto de detección.

Producciones de errores

Al anticipar los errores comunes que podríamos encontrar, podemos aumentar la gramática para el lenguaje, con producciones que generen las construcciones erróneas. Un analizador sintáctico construido a partir de una gramática aumentada por estas producciones de errores detecta los errores anticipados cuando se utiliza una producción de error durante el análisis sintáctico. Así, el analizador sintáctico puede generar diagnósticos de error apropiados sobre la construcción errónea que se haya reconocido en la entrada.

Corrección global

Lo ideal sería que un compilador hiciera la menor cantidad de cambios en el procesamiento de una cadena de entrada incorrecta. Hay algoritmos para elegir una secuencia mínima de cambios, para obtener una corrección con el menor costo a nivel global. Dada una cadena de entrada incorrecta x y una gramática G, estos algoritmos buscarán un árbol de análisis sintáctico para una cadena y relacionada, de tal forma que el número de inserciones, eliminaciones y modificaciones de los tokens requeridos para transformar a x en y sea lo más pequeño posible. Por desgracia, estos métodos son en general demasiado costosos para implementarlos en términos de tiempo y espacio, por lo cual estas técnicas sólo son de interés teórico en estos momentos.

Hay que observar que un programa casi correcto tal vez no sea lo que el programador tenía en mente. Sin embargo, la noción de la corrección con el menor costo proporciona una norma para evaluar las técnicas de recuperación de los errores, la cual se ha utilizado para buscar cadenas de sustitución óptimas para la recuperación a nivel de frase.

4.2 Gramáticas libres de contexto

En la sección 2.2 se presentaron las gramáticas para describir en forma sistemática la sintaxis de las construcciones de un lenguaje de programación, como las expresiones y las instrucciones. Si utilizamos una variable sintáctica *instr* para denotar las instrucciones, y una variable *expr* para denotar las expresiones, la siguiente producción:

$$\textit{instr} \quad \rightarrow \quad \textbf{if} \ (\ \textit{expr}\ )\ \textit{instr}\ \textbf{else}\ \textit{instr} \tag{4.4}$$

especifica la estructura de esta forma de instrucción condicional. Entonces, otras producciones definen con precisión lo que es una *expr* y qué más puede ser una *instr*.

En esta sección repasaremos la definición de una gramática libre de contexto y presentaremos la terminología para hablar acerca del análisis sintáctico. En especial, la noción de derivaciones es muy útil para hablar sobre el orden en el que se aplican las producciones durante el análisis sintáctico.

4.2.1 La definición formal de una gramática libre de contexto

En la sección 2.2 vimos que una gramática libre de contexto (o simplemente gramática) consiste en terminales, no terminales, un símbolo inicial y producciones.

1. Los *terminales* son los símbolos básicos a partir de los cuales se forman las cadenas. El término "nombre de token" es un sinónimo de "terminal"; con frecuencia usaremos la palabra "token" en vez de terminal, cuando esté claro que estamos hablando sólo sobre el nombre del token. Asumimos que las terminales son los primeros componentes de los tokens que produce el analizador léxico. En (4.4), los terminales son las palabras reservadas **if** y **else**, y los símbolos "`(`" y "`)`".

2. Los *no terminales* son variables sintácticas que denotan conjuntos de cadenas. En (4.4), *instr* y *expr* son no terminales. Los conjuntos de cadenas denotados por los no terminales ayudan a definir el lenguaje generado por la gramática. Los no terminales imponen una estructura jerárquica sobre el lenguaje, que representa la clave para el análisis sintáctico y la traducción.

3. En una gramática, un no terminal se distingue como el *símbolo inicial*, y el conjunto de cadenas que denota es el lenguaje generado por la gramática. Por convención, las producciones para el símbolo inicial se listan primero.

4. Las producciones de una gramática especifican la forma en que pueden combinarse los terminales y los no terminales para formar cadenas. Cada *producción* consiste en:

   (a) Un no terminal, conocido como *encabezado* o *lado izquierdo* de la producción; esta producción define algunas de las cadenas denotadas por el encabezado.

   (b) El símbolo $\rightarrow$. Algunas veces se ha utilizado `::=` en vez de la flecha.

   (c) Un *cuerpo* o *lado derecho*, que consiste en cero o más terminales y no terminales. Los componentes del cuerpo describen una forma en que pueden construirse las cadenas del no terminal en el encabezado.

Ejemplo 4.5: La gramática en la figura 4.2 define expresiones aritméticas simples. En esta gramática, los símbolos de los terminales son:

id + - * / ( )

Los símbolos de los no terminales son *expresión*, *term* y *factor*, y *expresión* es el símbolo inicial. □

$$
\begin{array}{rcl}
\textit{expresión} & \rightarrow & \textit{expresión} \texttt{ + } \textit{term} \\
\textit{expresión} & \rightarrow & \textit{expresión} \texttt{ - } \textit{term} \\
\textit{expresión} & \rightarrow & \textit{term} \\
\textit{term} & \rightarrow & \textit{term} \texttt{ * } \textit{factor} \\
\textit{term} & \rightarrow & \textit{term} \texttt{ / } \textit{factor} \\
\textit{term} & \rightarrow & \textit{factor} \\
\textit{factor} & \rightarrow & \texttt{(} \textit{expresión} \texttt{)} \\
\textit{factor} & \rightarrow & \textbf{id}
\end{array}
$$

Figura 4.2: Gramática para las expresiones aritméticas simples

4.2.2 Convenciones de notación

Para evitar siempre tener que decir que "éstos son los terminales", "éstos son los no terminales", etcétera, utilizaremos las siguientes convenciones de notación para las gramáticas durante el resto de este libro:

1. Estos símbolos son terminales:
   (a) Las primeras letras minúsculas del alfabeto, como *a*, *b*, *c*.
   (b) Los símbolos de operadores como +, ∗ , etcétera.
   (c) Los símbolos de puntuación como paréntesis, coma, etcétera.
   (d) Los dígitos 0, 1, ... , 9.
   (e) Las cadenas en negrita como **id** o **if**, cada una de las cuales representa un solo símbolo terminal.
2. Estos símbolos son no terminales:
   (a) Las primeras letras mayúsculas del alfabeto, como *A*, *B*, *C*.
   (b) La letra *S* que, al aparecer es, por lo general, el símbolo inicial.
   (c) Los nombres en cursiva y minúsculas, como *expr* o *instr*.
   (d) Al hablar sobre las construcciones de programación, las letras mayúsculas pueden utilizarse para representar no terminales. Por ejemplo, los no terminales para las expresiones, los términos y los factores se representan a menudo mediante *E*, *T* y *F*, respectivamente.

3. Las últimas letras mayúsculas del alfabeto, como X, Y, Z, representan *símbolos gramaticales*; es decir, pueden ser no terminales o terminales.
4. Las últimas letras minúsculas del alfabeto, como u, v, ..., z, representan cadenas de terminales (posiblemente vacías).
5. Las letras griegas minúsculas α, β, γ, por ejemplo, representan cadenas (posiblemente vacías) de símbolos gramaticales. Por ende, una producción genérica puede escribirse como $A \rightarrow \alpha$, en donde A es el encabezado y α el cuerpo.
6. Un conjunto de producciones $A \rightarrow \alpha_1$, $A \rightarrow \alpha_2$, ..., $A \rightarrow \alpha_k$ con un encabezado común A (las llamaremos *producciones* A), puede escribirse como $A \rightarrow \alpha_1 \mid \alpha_2 \mid \cdots \mid \alpha_k$. A α_1, α_2, ..., α_k les llamamos las *alternativas* para A.
7. A menos que se indique lo contrario, el encabezado de la primera producción es el símbolo inicial.

Ejemplo 4.6: Mediante estas convenciones, la gramática del ejemplo 4.5 puede rescribirse en forma concisa como:

$$\begin{array}{lcl} E & \rightarrow & E + T \mid E - T \mid T \\ T & \rightarrow & T * F \mid T / F \mid F \\ F & \rightarrow & (E) \mid \mathbf{id} \end{array}$$

Las convenciones de notación nos indican que E, T y F son no terminales, y E es el símbolo inicial. El resto de los símbolos son terminales. □

4.2.3 Derivaciones

La construcción de un árbol de análisis sintáctico puede hacerse precisa si tomamos una vista derivacional, en la cual las producciones se tratan como reglas de rescritura. Empezando con el símbolo inicial, cada paso de rescritura sustituye a un no terminal por el cuerpo de una de sus producciones. Esta vista derivacional corresponde a la construcción descendente de un árbol de análisis sintáctico, pero la precisión que ofrecen las derivaciones será muy útil cuando hablemos del análisis sintáctico ascendente. Como veremos, el análisis sintáctico ascendente se relaciona con una clase de derivaciones conocidas como derivaciones de "más a la derecha", en donde el no terminal por la derecha se rescribe en cada paso.

Por ejemplo, considere la siguiente gramática, con un solo no terminal E, la cual agrega una producción $E \rightarrow - E$ a la gramática (4.3):

$$E \rightarrow E + E \mid E * E \mid - E \mid (E) \mid \mathbf{id} \tag{4.7}$$

La producción $E \rightarrow - E$ significa que si E denota una expresión, entonces $- E$ debe también denotar una expresión. La sustitución de una sola E por $- E$ se describirá escribiendo lo siguiente:

$$E \Rightarrow -E$$

lo cual se lee como "E deriva a $-\ E$". La producción $E \rightarrow (\ E\ )$ puede aplicarse para sustituir cualquier instancia de E en cualquier cadena de símbolos gramaticales por (E); por ejemplo, $E * E \Rightarrow (E) * E$ o $E * E \Rightarrow E * (E)$. Podemos tomar una sola E y aplicar producciones en forma repetida y en cualquier orden para obtener una secuencia de sustituciones. Por ejemplo,

$$E \Rightarrow -E \Rightarrow -(E) \Rightarrow -(\mathbf{id})$$

A dicha secuencia de sustituciones la llamamos una *derivación* de $-(\mathbf{id})$ a partir de E. Esta derivación proporciona la prueba de que la cadena $-(\mathbf{id})$ es una instancia específica de una expresión.

Para una definición general de la derivación, considere un no terminal A en la mitad de una secuencia de símbolos gramaticales, como en $\alpha A \beta$, en donde α y β son cadenas arbitrarias de símbolos gramaticales. Suponga que $A \rightarrow \gamma$ es una producción. Entonces, escribimos $\alpha A \beta \Rightarrow \alpha\gamma\beta$. El símbolo $\Rightarrow$ significa, "se deriva en un paso". Cuando una secuencia de pasos de derivación $\alpha_1 \Rightarrow \alpha_2 \Rightarrow \cdots \Rightarrow \alpha_n$ se rescribe como α_1 a α_n, decimos que α_1 *deriva* a α_n. Con frecuencia es conveniente poder decir, "deriva en cero o más pasos". Para este fin, podemos usar el símbolo $\overset{*}{\Rightarrow}$. Así,

1. $\alpha \overset{*}{\Rightarrow} \alpha$, para cualquier cadena α.

2. Si $\alpha \overset{*}{\Rightarrow} \beta$ y $\beta \Rightarrow \gamma$, entonces $\alpha \overset{*}{\Rightarrow} \gamma$.

De igual forma, $\overset{+}{\Rightarrow}$ significa "deriva en uno o más pasos".

Si $S \overset{*}{\Rightarrow} \alpha$, en donde S es el símbolo inicial de una gramática G, decimos que α es una *forma de frase* de G. Observe que una forma de frase puede contener tanto terminales como no terminales, y puede estar vacía. Un *enunciado* de G es una forma de frase sin símbolos no terminales. El *lenguaje generado* por una gramática es su conjunto de oraciones. Por ende, una cadena de terminales w está en $L(G)$, el lenguaje generado por G, si y sólo si w es un enunciado de G (o $S \overset{*}{\Rightarrow} w$). Un lenguaje que puede generarse mediante una gramática se considera un *lenguaje libre de contexto*. Si dos gramáticas generan el mismo lenguaje, se consideran como *equivalentes*.

La cadena $-(\mathbf{id} + \mathbf{id})$ es un enunciado de la gramática (4.7), ya que hay una derivación

$$E \Rightarrow -E \Rightarrow -(E) \Rightarrow -(E + E) \Rightarrow -(\mathbf{id} + E) \Rightarrow -(\mathbf{id} + \mathbf{id}) \tag{4.8}$$

Las cadenas E, $-E$, $-(E)$, ..., $-(\mathbf{id} + \mathbf{id})$ son todas formas de frases de esta gramática. Escribimos $E \overset{*}{\Rightarrow} -(\mathbf{id} + \mathbf{id})$ para indicar que $-(\mathbf{id} + \mathbf{id})$ puede derivarse de E.

En cada paso de una derivación, hay dos elecciones por hacer. Debemos elegir qué no terminal debemos sustituir, y habiendo realizado esta elección, debemos elegir una producción con ese no terminal como encabezado. Por ejemplo, la siguiente derivación alternativa de $-(\mathbf{id} + \mathbf{id})$ difiere de la derivación (4.8) en los últimos dos pasos:

$$E \Rightarrow -E \Rightarrow -(E) \Rightarrow -(E + E) \Rightarrow -(E + \mathbf{id}) \Rightarrow -(\mathbf{id} + \mathbf{id}) \tag{4.9}$$

Cada no terminal se sustituye por el mismo cuerpo en las dos derivaciones, pero el orden de las sustituciones es distinto.

Para comprender la forma en que trabajan los analizadores sintácticos, debemos considerar las derivaciones en las que el no terminal que se va a sustituir en cada paso se elige de la siguiente manera:

1. En las derivaciones *por la izquierda*, siempre se elige el no terminal por la izquierda en cada de frase. Si $\alpha \Rightarrow \beta$ es un paso en el que se sustituye el no terminal por la izquierda en α, escribimos $\alpha \underset{lm}{\Rightarrow} \beta$.
2. En las derivaciones *por la derecha*, siempre se elige el no terminal por la derecha; en este caso escribimos $\alpha \underset{rm}{\Rightarrow} \beta$.

La derivación (4.8) es por la izquierda, por lo que puede rescribirse de la siguiente manera:

$$E \underset{lm}{\Rightarrow} -E \underset{lm}{\Rightarrow} -(E) \underset{lm}{\Rightarrow} -(E + E) \underset{lm}{\Rightarrow} -(\mathbf{id} + E) \underset{lm}{\Rightarrow} -(\mathbf{id} + \mathbf{id})$$

Observe que (4.9) es una derivación por la derecha.

Si utilizamos nuestras convenciones de notación, cada paso por la izquierda puede escribirse como $wA\gamma \underset{lm}{\Rightarrow} w\delta\gamma$, en donde w consiste sólo de terminales, $A \rightarrow \delta$ es la producción que se aplica, y γ es una cadena de símbolos gramaticales. Para enfatizar que α deriva a β mediante una derivación por la izquierda, escribimos $\alpha \overset{*}{\underset{lm}{\Rightarrow}} \beta$. Si $S \overset{*}{\underset{lm}{\Rightarrow}} \alpha$, decimos que α es una *forma de frase izquierda* de la gramática en cuestión.

Las análogas definiciones son válidas para las derivaciones por la derecha. A estas derivaciones se les conoce algunas veces como derivaciones *canónicas*.

4.2.4 Árboles de análisis sintáctico y derivaciones

Un árbol de análisis sintáctico es una representación gráfica de una derivación que filtra el orden en el que se aplican las producciones para sustituir los no terminales. Cada nodo interior de un árbol de análisis sintáctico representa la aplicación de una producción. El nodo interior se etiqueta con el no terminal A en el encabezado de la producción; los hijos del nodo se etiquetan, de izquierda a derecha, mediante los símbolos en el cuerpo de la producción por la que se sustituyó esta A durante la derivación.

Por ejemplo, el árbol de análisis sintáctico para $-(\mathbf{id} + \mathbf{id})$ en la figura 4.3 resulta de la derivación (4.8), así como de la derivación (4.9).

Las hojas de un árbol de análisis sintáctico se etiquetan mediante no terminales o terminales y, leídas de izquierda a derecha, constituyen una forma de frase, a la cual se le llama *producto* o *frontera* del árbol.

Para ver la relación entre las derivaciones y los árboles de análisis sintáctico, considere cualquier derivación $\alpha_1 \Rightarrow \alpha_2 \Rightarrow \cdots \Rightarrow \alpha_n$, en donde α_1 es un sólo no terminal A. Para cada forma de frase α_i en la derivación, podemos construir un árbol de análisis sintáctico cuyo producto sea α_i. El proceso es una inducción sobre i.

BASE: El árbol para $\alpha_1 = A$ es un solo nodo, etiquetado como A.

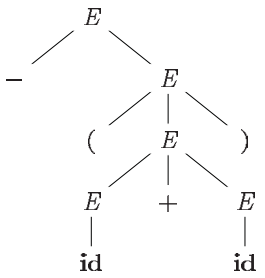

Figura 4.3: Árbol de análisis sintáctico para $-(\mathbf{id} + \mathbf{id})$

INDUCCIÓN: Suponga que ya hemos construido un árbol de análisis sintáctico con el producto $\alpha_{i-1} = X_1 X_2 \cdots X_k$ (tenga en cuenta que, de acuerdo a nuestras convenciones de notación, cada símbolo gramatical X_i es un no terminal o un terminal). Suponga que α_i se deriva de α_{i-1} al sustituir X_j, un no terminal, por $\beta = Y_1 Y_2 \cdots Y_m$. Es decir, en el i-ésimo paso de la derivación, la producción $X_j \rightarrow \beta$ se aplica a α_{i-1} para derivar $\alpha_i = X_1 X_2 \cdots X_{j-1} \beta X_{j+1} \cdots X_k$.

Para modelar este paso de la derivación, buscamos la j-ésima hoja, partiendo de la izquierda en el árbol de análisis sintáctico actual. Esta hoja se etiqueta como X_j. A esta hoja le damos m hijos, etiquetados $Y_1, Y_2, \ldots, Y_m$, partiendo de la izquierda. Como caso especial, si $m = 0$ entonces $\beta = \epsilon$, y proporcionamos a la j-ésima hoja un hijo etiquetado como ϵ.

Ejemplo 4.10: La secuencia de árboles de análisis sintáctico que se construyen a partir de la derivación (4.8) se muestra en la figura 4.4. En el primer paso de la derivación, $E \Rightarrow -E$. Para modelar este paso, se agregan dos hijos, etiquetados como $-$ y E, a la raíz E del árbol inicial. El resultado es el segundo árbol.

En el segundo paso de la derivación, $-E \Rightarrow -(E)$. Por consiguiente, agregamos tres hijos, etiquetados como `(`, E y `)`, al nodo hoja etiquetado como E del segundo árbol, para obtener el tercer árbol con coséchale producto $-(E)$. Si continuamos de esta forma, obtenemos el árbol de análisis sintáctico completo como el sexto árbol. □

Como un árbol de análisis sintáctico ignora las variaciones en el orden en el que se sustituyen los símbolos en las formas de las oraciones, hay una relación de varios a uno entre las derivaciones y los árboles de análisis sintáctico. Por ejemplo, ambas derivaciones (4.8) y (4.9) se asocian con el mismo árbol de análisis sintáctico final de la figura 4.4.

En lo que sigue, realizaremos con frecuencia el análisis sintáctico produciendo una derivación por la izquierda o por la derecha, ya que hay una relación de uno a uno entre los árboles de análisis sintáctico y este tipo de derivaciones. Tanto las derivaciones por la izquierda como las de por la derecha eligen un orden específico para sustituir símbolos en las formas de las oraciones, por lo que también filtran las variaciones en orden. No es difícil mostrar que todos los árboles sintácticos tienen asociadas una derivación única por la izquierda y una derivación única por la derecha.

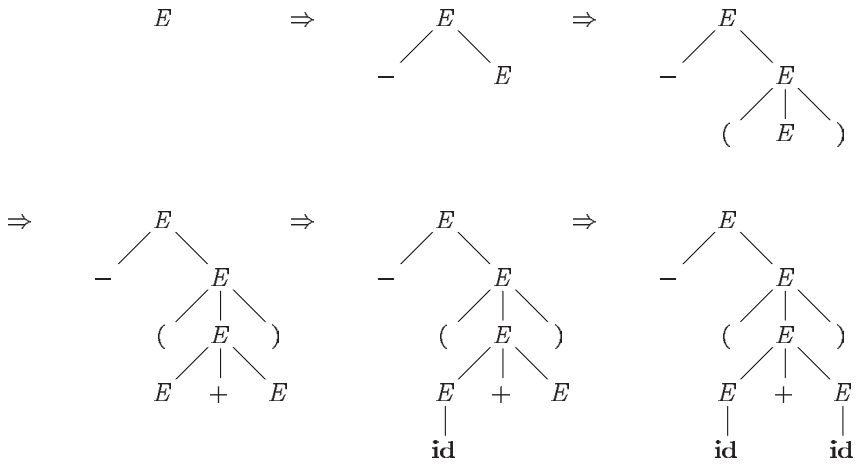

Figura 4.4: Secuencia de árboles de análisis sintáctico para la derivación (4.8)

4.2.5 Ambigüedad

En la sección 2.2.4 vimos que una gramática que produce más de un árbol de análisis sintáctico para cierto enunciado es *ambiguo*. Dicho de otra forma, una gramática ambigua es aquella que produce más de una derivación por la izquierda, o más de una derivación por la derecha para el mismo enunciado.

Ejemplo 4.11: La gramática de expresiones aritméticas (4.3) permite dos derivaciones por la izquierda distintas para el enunciado **id** + **id** ∗ **id**:

$$
\begin{array}{rcl@{\qquad}rcl}
E & \Rightarrow & E+E & E & \Rightarrow & E*E \\
 & \Rightarrow & \mathbf{id}+E & & \Rightarrow & E+E*E \\
 & \Rightarrow & \mathbf{id}+E*E & & \Rightarrow & \mathbf{id}+E*E \\
 & \Rightarrow & \mathbf{id}+\mathbf{id}*E & & \Rightarrow & \mathbf{id}+\mathbf{id}*E \\
 & \Rightarrow & \mathbf{id}+\mathbf{id}*\mathbf{id} & & \Rightarrow & \mathbf{id}+\mathbf{id}*\mathbf{id}
\end{array}
$$

Los árboles de análisis sintáctico correspondientes aparecen en la figura 4.5.

Observe que el árbol de análisis sintáctico de la figura 4.5(a) refleja la precedencia que se asume comúnmente para + y ∗, mientras que el árbol de la figura 4.5(b) no. Es decir, lo común es tratar al operador ∗ teniendo mayor precedencia que +, en forma correspondiente al hecho de que, por lo general, evaluamos la expresión $a + b * c$ como $a + (b * c)$, en vez de hacerlo como $(a + b) * c$. □

Para la mayoría de los analizadores sintácticos, es conveniente que la gramática no tenga ambigüedades, ya que de lo contrario, no podemos determinar en forma única qué árbol de análisis sintáctico seleccionar para un enunciado. En otros casos, es conveniente usar gramáticas ambiguas elegidas con cuidado, junto con *reglas para eliminar la ambigüedad*, las cuales "descartan" los árboles sintácticos no deseados, dejando sólo un árbol para cada enunciado.

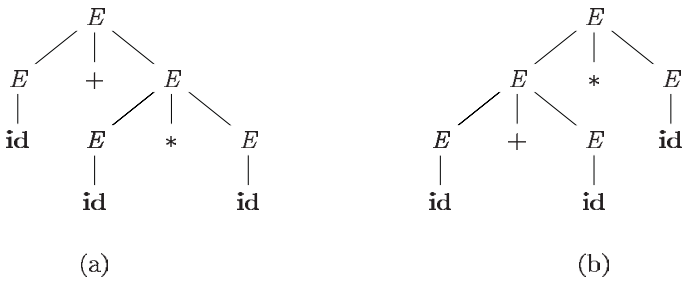

Figura 4.5: Dos árboles de análisis sintáctico para **id+id*id**

4.2.6 Verificación del lenguaje generado por una gramática

Aunque los diseñadores de compiladores muy raras veces lo hacen para una gramática de lenguaje de programación completa, es útil poder razonar que un conjunto dado de producciones genera un lenguaje específico. Las construcciones problemáticas pueden estudiarse mediante la escritura de una gramática abstracta y concisa, y estudiando el lenguaje que genera. A continuación vamos a construir una gramática de este tipo, para instrucciones condicionales.

Una prueba de que una gramática G genera un lenguaje L consta de dos partes: mostrar que todas las cadenas generadas por G están en L y, de manera inversa, que todas las cadenas en L pueden generarse sin duda mediante G.

Ejemplo 4.12: Considere la siguiente gramática:

$$S \rightarrow (\ S \ ) \ S \mid \epsilon \tag{4.13}$$

Tal vez no sea evidente desde un principio, pero esta gramática simple genera todas las cadenas de paréntesis balanceados, y sólo ese tipo de cadenas. Para ver por qué, primero mostraremos que todas las frases que se derivan de S son balanceadas, y después que todas las cadenas balanceadas se derivan de S. Para mostrar que todas las frases que pueden derivarse de S son balanceadas, utilizaremos una prueba inductiva sobre el número de pasos n en una derivación.

BASE: La base es $n = 1$. La única cadena de terminales que puede derivarse de S en un paso es la cadena vacía, que sin duda está balanceada.

INDUCCIÓN: Ahora suponga que todas las derivaciones de menos de n pasos producen frases balanceadas, y considere una derivación por la izquierda, con n pasos exactamente. Dicha derivación debe ser de la siguiente forma:

$$S \underset{lm}{\Rightarrow} (S)S \overset{*}{\underset{lm}{\Rightarrow}} (x)S \overset{*}{\underset{lm}{\Rightarrow}} (x)y$$

Las derivaciones de x y y que provienen de S requieren menos de n pasos, por lo que en base a la hipótesis inductiva, x y y están balanceadas. Por lo tanto, la cadena $(x)y$ debe ser balanceada. Es decir, tiene un número equivalente de paréntesis izquierdos y derechos, y cada prefijo tiene, por lo menos, la misma cantidad de paréntesis izquierdos que derechos.

Habiendo demostrado entonces que cualquier cadena que se deriva de S está balanceada, debemos ahora mostrar que todas las cadenas balanceadas se derivan de S. Para ello, utilizaremos la inducción sobre la longitud de una cadena.

BASE: Si la cadena es de longitud 0, debe ser ϵ, la cual está balanceada.

INDUCCIÓN: Primero, observe que todas las cadenas balanceadas tienen longitud uniforme. Suponga que todas las cadenas balanceadas de una longitud menor a $2n$ se derivan de S, y considere una cadena balanceada w de longitud $2n$, $n \geq 1$. Sin duda, w empieza con un paréntesis izquierdo. Hagamos que (x) sea el prefijo no vacío más corto de w, que tenga el mismo número de paréntesis izquierdos y derechos. Así, w puede escribirse como $w = (x)y$, en donde x y y están balanceadas. Como x y y tienen una longitud menor a $2n$, pueden derivarse de S mediante la hipótesis inductiva. Por ende, podemos buscar una derivación de la siguiente forma:

$$S \Rightarrow (S)S \stackrel{*}{\Rightarrow} (x)S \stackrel{*}{\Rightarrow} (x)y$$

con lo cual demostramos que $w = (x)y$ también puede derivarse de S. □

4.2.7 Comparación entre gramáticas libres de contexto y expresiones regulares

Antes de dejar esta sección sobre las gramáticas y sus propiedades, establecemos que las gramáticas son una notación más poderosa que las expresiones regulares. Cada construcción que puede describirse mediante una expresión regular puede describirse mediante una gramática, pero no al revés. De manera alternativa, cada lenguaje regular es un lenguaje libre de contexto, pero no al revés.

Por ejemplo, la expresión regular $(\mathbf{a}|\mathbf{b})^*\mathbf{abb}$ y la siguiente gramática:

$$\begin{array}{lcl} A_0 & \rightarrow & aA_0 \mid bA_0 \mid aA_1 \\ A_1 & \rightarrow & bA_2 \\ A_2 & \rightarrow & bA_3 \\ A_3 & \rightarrow & \epsilon \end{array}$$

describen el mismo lenguaje, el conjunto de cadenas de as y bs que terminan en abb.

Podemos construir de manera mecánica una gramática para reconocer el mismo lenguaje que un autómata finito no determinista (AFN). La gramática anterior se construyó a partir del AFN de la figura 3.24, mediante la siguiente construcción:

1. Para cada estado i del AFN, crear un no terminal A_i.
2. Si el estado i tiene una transición al estado j con la entrada a, agregar la producción $A_i \rightarrow aA_j$. Si el estado i pasa al estado j con la entrada ϵ, agregar la producción $A_i \rightarrow A_j$.
3. Si i es un estado de aceptación, agregar $A_i \rightarrow \epsilon$.
4. Si i es el estado inicial, hacer que A_i sea el símbolo inicial de la gramática.

Por otra parte, el lenguaje $L = \{a^n b^n \mid n \geq 1\}$ con un número equivalente de as y bs es un ejemplo de prototipo de un lenguaje que puede describirse mediante una gramática, pero no mediante una expresión regular. Para ver por qué, suponga que L es el lenguaje definido por alguna expresión regular. Construiríamos un AFD D con un número finito de estados, por decir k, para aceptar a L. Como D sólo tiene k estados, para una entrada que empieza con más de k as, D debe entrar a cierto estado dos veces, por decir s_i, como en la figura 4.6. Suponga que la ruta de s_i de vuelta a sí mismo se etiqueta con una secuencia a^{j-i}. Como $a^i b^i$ está en el lenguaje, debe haber una ruta etiquetada como b^i desde s_i hasta un estado de aceptación f. Pero, entonces también hay una ruta que sale desde el estado s_0 y pasa a través de s_i para llegar a f, etiquetada como $a^j b^i$, como se muestra en la figura 4.6. Por ende, D también acepta a $a^j b^i$, que no está en el lenguaje, lo cual contradice la suposición de que L es el lenguaje aceptado por D.

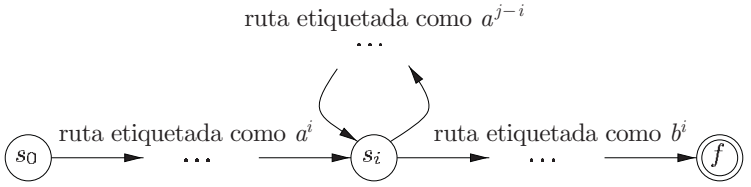

Figura 4.6: Un AFD D que acepta a $a^i b^i$ y a $a^j b^i$

En lenguaje coloquial, decimos que "los autómatas finitos no pueden contar", lo cual significa que un autómata finito no puede aceptar un lenguaje como $\{a^n b^n \mid n \geq 1\}$, que requiera que el autómata lleve la cuenta del número de as antes de ver las bs. De igual forma, "una gramática puede contar dos elementos pero no tres", como veremos cuando hablemos sobre las construcciones de lenguajes que no son libres de contexto en la sección 4.3.5.

4.2.8 Ejercicios para la sección 4.2

Ejercicio 4.2.1: Considere la siguiente gramática libre de contexto:

$$S \rightarrow S\ S + \mid S\ S * \mid a$$

y la cadena $aa + a*$.

a) Proporcione una derivación por la izquierda para la cadena.

b) Proporcione una derivación por la derecha para la cadena.

c) Proporcione un árbol de análisis sintáctico para la cadena.

! d) ¿La gramática es ambigua o no? Justifique su respuesta.

! e) Describa el lenguaje generado por esta gramática.

Ejercicio 4.2.2: Repita el ejercicio 4.2.1 para cada una de las siguientes gramáticas y cadenas:

a) $S \rightarrow 0\ S\ 1 \mid 0\ 1$ con la cadena 000111.

b) $S \rightarrow +\ S\ S \mid *\ S\ S \mid a$ con la cadena $+\ *\ aaa$.

! c) $S \rightarrow S\ (\ S\ )\ S \mid \epsilon$ con la cadena (()()).

! d) $S \rightarrow S + S \mid S\ S \mid (\ S\ ) \mid S * \mid a$ con la cadena $(a + a) * a$.

! e) $S \rightarrow (\ L\ ) \mid a$ y $L \rightarrow L,\ S \mid S$ con la cadena $((a,\ a),\ a,\ (a))$.

!! f) $S \rightarrow a\ S\ b\ S \mid b\ S\ a\ S \mid \epsilon$ con la cadena *aabbab*.

! g) La siguiente gramática para las expresiones booleanas:

$$
\begin{array}{lcl}
\textit{bexpr} & \rightarrow & \textit{bexpr}\ \textbf{or}\ \textit{bterm} \mid \textit{bterm} \\
\textit{bterm} & \rightarrow & \textit{bterm}\ \textbf{and}\ \textit{bfactor} \mid \textit{bfactor} \\
\textit{bfactor} & \rightarrow & \textbf{not}\ \textit{bfactor} \mid (\ \textit{bexpr}\ ) \mid \textbf{true} \mid \textbf{false}
\end{array}
$$

Ejercicio 4.2.3: Diseñe gramáticas para los siguientes lenguajes:

a) El conjunto de todas las cadenas de 0s y 1s, de tal forma que justo antes de cada 0 vaya por lo menos un 1.

! b) El conjunto de todas las cadenas de 0s y 1s que sean *palíndromos*; es decir, que la cadena se lea igual al derecho y al revés.

! c) El conjunto de todas las cadenas de 0s y 1s con un número igual de 0s y 1s.

!! d) El conjunto de todas las cadenas de 0s y 1s con un número desigual de 0s y 1s.

! e) El conjunto de todas las cadenas de 0s y 1s en donde 011 no aparece como una subcadena.

!! f) El conjunto de todas las cadenas de 0s y 1s de la forma xy, en donde $x \neq y$, y x y y tienen la misma longitud.

! **Ejercicio 4.2.4:** Hay una notación de gramática extendida de uso común. En esta notación, los corchetes y las llaves en los cuerpos de las producciones son meta símbolos (como $\rightarrow$ o $|$) con los siguientes significados:

i) Los corchetes alrededor de un símbolo o símbolos gramaticales denota que estas construcciones son opcionales. Por ende, la producción $A \rightarrow X\ [\,Y\,]\ Z$ tiene el mismo efecto que las dos producciones $A \rightarrow X\ Y\ Z$ y $A \rightarrow X\ Z$.

ii) Las llaves alrededor de un símbolo o símbolos gramaticales indican que estos símbolos pueden repetirse cualquier número de veces, incluyendo cero. Por ende, $A \rightarrow X\ \{\,Y\ Z\}$ tiene el mismo efecto que la secuencia infinita de producciones $A \rightarrow X$, $A \rightarrow X\ Y\ Z$, $A \rightarrow X\ Y\ Z\ Y\ Z$, y así sucesivamente.

Muestre que estas dos extensiones no agregan potencia a las gramáticas; es decir, cualquier lenguaje que pueda generarse mediante una gramática con estas extensiones, podrá generarse mediante una gramática sin las extensiones.

Ejercicio 4.2.5: Use las llaves descritas en el ejercicio 4.2.4 para simplificar la siguiente gramática para los bloques de instrucciones y las instrucciones condicionales:

$$
\begin{array}{lcl}
\textit{instr} & \rightarrow & \textbf{if } \textit{expr} \textbf{ then } \textit{instr} \textbf{ else } \textit{instr} \\
& | & \textbf{if } \textit{instr} \textbf{ then } \textit{instr} \\
& | & \textbf{begin } \textit{listaInstr} \textbf{ end} \\
\textit{listaInstr} & \rightarrow & \textit{instr} \textbf{ ; } \textit{listaInstr} \mid \textit{instr}
\end{array}
$$

! **Ejercicio 4.2.6:** Extienda la idea del ejercicio 4.2.4 para permitir cualquier expresión regular de símbolos gramaticales en el cuerpo de una producción. Muestre que esta extensión no permite que las gramáticas definan nuevos lenguajes.

! **Ejercicio 4.2.7:** Un símbolo gramatical X (terminal o no terminal) es *inútil* si no hay derivación de la forma $S \stackrel{*}{\Rightarrow} wXy \stackrel{*}{\Rightarrow} wxy$. Es decir, X nunca podrá aparecer en la derivación de un enunciado.

a) Proporcione un algoritmo para eliminar de una gramática todas las producciones que contengan símbolos inútiles.

b) Aplique su algoritmo a la siguiente gramática:

$$
\begin{array}{lcl}
S & \rightarrow & 0 \mid A \\
A & \rightarrow & AB \\
B & \rightarrow & 1
\end{array}
$$

Ejercicio 4.2.8: La gramática en la figura 4.7 genera declaraciones para un solo identificador numérico; estas declaraciones involucran a cuatro propiedades distintas e independientes de números.

$$
\begin{array}{lcl}
\textit{instr} & \rightarrow & \textbf{declare id } \textit{listaOpciones} \\
\textit{listaOpciones} & \rightarrow & \textit{listaOpciones opcion} \mid \epsilon \\
\textit{opcion} & \rightarrow & \textit{modo} \mid \textit{escala} \mid \textit{precision} \mid \textit{base} \\
\textit{modo} & \rightarrow & \textbf{real} \mid \textbf{complex} \\
\textit{escala} & \rightarrow & \textbf{fixed} \mid \textbf{floating} \\
\textit{precision} & \rightarrow & \textbf{single} \mid \textbf{double} \\
\textit{base} & \rightarrow & \textbf{binary} \mid \textbf{decimal}
\end{array}
$$

Figura 4.7: Una gramática para declaraciones con varios atributos

a) Generalice la gramática de la figura 4.7, permitiendo n opciones A_i, para algunas n fijas y para $i = 1, 2, \ldots, n$, en donde A_i puede ser a_i o b_i. Su gramática deberá usar sólo $O(n)$ símbolos gramaticales y tener una longitud total de producciones igual a $O(n)$.

! b) La gramática de la figura 4.7 y su generalización en la parte (a) permite declaraciones que son contradictorias o redundantes, tales como:

```
declare foo real fixed real floating
```

Podríamos insistir en que la sintaxis del lenguaje prohíbe dichas declaraciones; es decir, cada declaración generada por la gramática tiene exactamente un valor para cada una de las n opciones. Si lo hacemos, entonces para cualquier n fija hay sólo un número finito de declaraciones legales. Por ende, el lenguaje de declaraciones legales tiene una gramática (y también una expresión regular), al igual que cualquier lenguaje finito. La gramática obvia, en la cual el símbolo inicial tiene una producción para cada declaración legal, tiene $n!$ producciones y una longitud de producciones total de $O(n \times n!)$. Hay que esforzarse más: una longitud de producciones total que sea $O(n2^n)$.

!! c) Muestre que cualquier gramática para la parte (b) debe tener una longitud de producciones total de por lo menos 2^n.

d) ¿Qué dice la parte (c) acerca de la viabilidad de imponer la no redundancia y la no contradicción entre las opciones en las declaraciones, a través de la sintaxis del lenguaje de programación?

4.3 Escritura de una gramática

Las gramáticas son capaces de describir casi la mayoría de la sintaxis de los lenguajes de programación. Por ejemplo, el requerimiento de que los identificadores deben declararse antes de usarse, no puede describirse mediante una gramática libre de contexto. Por lo tanto, las secuencias de los tokens que acepta un analizador sintáctico forman un superconjunto del lenguaje de programación; las fases siguientes del compilador deben analizar la salida del analizador sintáctico, para asegurar que cumpla con las reglas que no verifica el analizador sintáctico.

Esta sección empieza con una discusión acerca de cómo dividir el trabajo entre un analizador léxico y un analizador sintáctico. Después consideraremos varias transformaciones que podrían aplicarse para obtener una gramática más adecuada para el análisis sintáctico. Una técnica puede eliminar la ambigüedad en la gramática, y las otras (eliminación de recursividad por la izquierda y factorización por la izquierda) son útiles para rescribir las gramáticas, de manera que sean adecuadas para el análisis sintáctico descendente. Concluiremos esta sección considerando algunas construcciones de los lenguajes de programación que ninguna gramática puede describir.

4.3.1 Comparación entre análisis léxico y análisis sintáctico

Como observamos en la sección 4.2.7, todo lo que puede describirse mediante una expresión regular también puede describirse mediante una gramática. Por lo tanto, sería razonable preguntar: "¿Por qué usar expresiones regulares para definir la sintaxis léxica de un lenguaje?" Existen varias razones.

1. Al separar la estructura sintáctica de un lenguaje en partes léxicas y no léxicas, se proporciona una manera conveniente de colocar en módulos la interfaz de usuario de un compilador en dos componentes de un tamaño manejable.
2. Las reglas léxicas de un lenguaje son con frecuencia bastante simples, y para describirlas no necesitamos una notación tan poderosa como las gramáticas.
3. Por lo general, las expresiones regulares proporcionan una notación más concisa y fácil de entender para los tokens, en comparación con las gramáticas.
4. Pueden construirse analizadores léxicos más eficientes en forma automática a partir de expresiones regulares, en comparación con las gramáticas arbitrarias.

No hay lineamientos firmes en lo que se debe poner en las reglas léxicas, en contraste a las reglas sintácticas. Las expresiones regulares son muy útiles para describir la estructura de las construcciones como los identificadores, las constantes, las palabras reservadas y el espacio en blanco. Por otro lado, las gramáticas son muy útiles para describir estructuras anidadas, como los paréntesis balanceados, las instrucciones begin-end relacionadas, las instrucciones if-then-else correspondientes, etcétera. Estas estructuras anidadas no pueden describirse mediante las expresiones regulares.

4.3.2 Eliminación de la ambigüedad

Algunas veces, una gramática ambigua puede rescribirse para eliminar la ambigüedad. Como ejemplo, vamos a eliminar la ambigüedad de la siguiente gramática del "else colgante":

$$\begin{array}{lcl} \textit{instr} & \rightarrow & \textbf{if } \textit{expr} \textbf{ then } \textit{instr} \\ & | & \textbf{if } \textit{expr} \textbf{ then } \textit{instr} \textbf{ else } \textit{instr} \\ & | & \textbf{otra} \end{array} \tag{4.14}$$

Aquí, "**otra**" representa a cualquier otra instrucción. De acuerdo con esta gramática, la siguiente instrucción condicional compuesta:

if E_1 **then** S_1 **else if** E_2 **then** S_2 **else** S_3

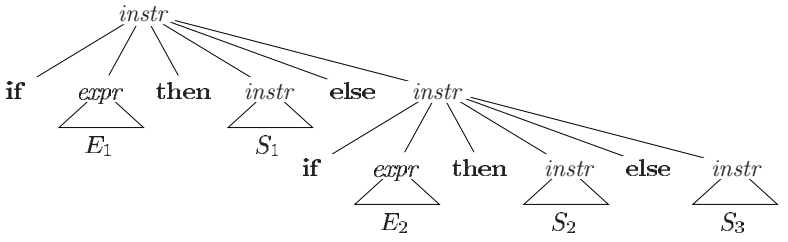

Figura 4.8: Árbol de análisis sintáctico para una instrucción condicional

tiene el árbol de análisis sintáctico que se muestra en la figura 4.8.[1] La gramática (4.14) es ambigua, ya que la cadena

$$\textbf{if } E_1 \textbf{ then if } E_2 \textbf{ then } S_1 \textbf{ else } S_2 \tag{4.15}$$

tiene los dos árboles de análisis sintáctico que se muestran en la figura 4.9.

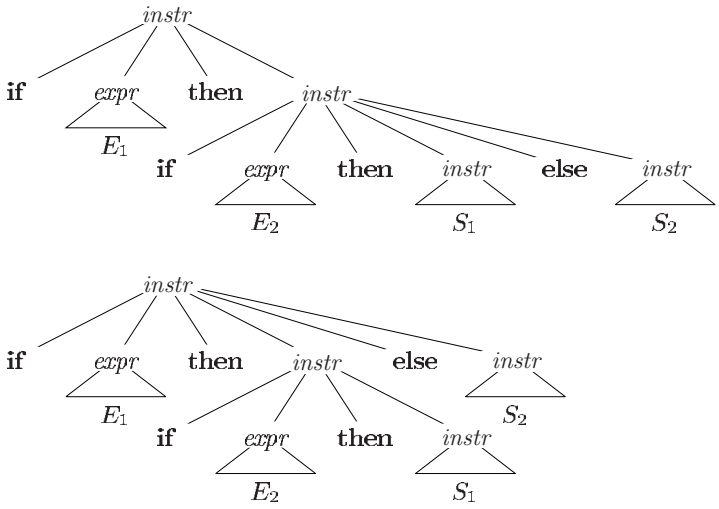

Figura 4.9: Dos árboles de análisis sintáctico para un enunciado ambiguo

En todos los lenguajes de programación con instrucciones condicionales de esta forma, se prefiere el primer árbol de análisis sintáctico. La regla general es, "Relacionar cada **else** con el **then** más cercano que no esté relacionado".[2] Esta regla para eliminar ambigüedad puede, en teoría, incorporarse directamente en una gramática, pero en la práctica raras veces se integra a las producciones.

Ejemplo 4.16: Podemos rescribir la gramática del else colgante (4.14) como la siguiente gramática sin ambigüedades. La idea es que una instrucción que aparece entre un **then** y un **else** debe estar "relacionada"; es decir, la instrucción interior no debe terminar con un **then** sin relacionar o abierto. Una instrucción relacionada es una instrucción **if-then-else** que no contiene instrucciones abiertas, o es cualquier otro tipo de instrucción incondicional. Por ende, podemos usar la gramática de la figura 4.10. Esta gramática genera las mismas cadenas que la gramática del else colgante (4.14), pero sólo permite un análisis sintáctico para la cadena (4.15); en específico, el que asocia a cada **else** con la instrucción **then** más cercana que no haya estado relacionada antes. □

[1] Los subíndices en E y S son sólo para diferenciar las distintas ocurrencias del mismo no terminal, por lo cual no implican no terminales distintos.

[2] Hay que tener en cuenta que C y sus derivados se incluyen en esta clase. Aun cuando la familia de lenguajes C no utiliza la palabra clave **then**, su función se representa mediante el paréntesis de cierre para la condición que va después de **if**.

$$
\begin{array}{rcl}
\textit{instr} & \rightarrow & \textit{instr\_relacionada} \\
& | & \textit{instr\_abierta} \\
\textit{instr\_relacionada} & \rightarrow & \textbf{if } \textit{expr} \textbf{ then } \textit{instr\_relacionada} \textbf{ else } \textit{instr\_relacionada} \\
& | & \textbf{otra} \\
\textit{instr\_abierta} & \rightarrow & \textbf{if } \textit{expr} \textbf{ then } \textit{instr} \\
& | & \textbf{if } \textit{expr} \textbf{ then } \textit{instr\_relacionada} \textbf{ else } \textit{instr\_abierta}
\end{array}
$$

Figura 4.10: Gramática sin ambigüedades para las instrucciones if-then-else

4.3.3 Eliminación de la recursividad por la izquierda

Una gramática es *recursiva por la izquierda* si tiene una terminal A tal que haya una derivación $A \stackrel{+}{\Rightarrow} A\alpha$ para cierta cadena α. Los métodos de análisis sintáctico descendentes no pueden manejar las gramáticas recursivas por la izquierda, por lo que se necesita una transformación para eliminar la recursividad por la izquierda. En la sección 2.4.5 hablamos sobre la *recursividad inmediata por la izquierda*, en donde hay una producción de la forma $A \rightarrow A\alpha$. Aquí estudiaremos el caso general. En la sección 2.4.5, mostramos cómo el par recursivo por la izquierda de producciones $A \rightarrow A\alpha \mid \beta$ podía sustituirse mediante las siguientes producciones no recursivas por la izquierda:

$$
\begin{array}{l}
A \rightarrow \beta A' \\
A' \rightarrow \alpha A' \mid \epsilon
\end{array}
$$

sin cambiar las cadenas que se derivan de A. Esta regla por sí sola basta para muchas gramáticas.

Ejemplo 4.17: La gramática de expresiones no recursivas por la izquierda (4.2), que se repite a continuación:

$$
\begin{array}{l}
E \rightarrow T\ E' \\
E' \rightarrow +\ T\ E' \\
T \rightarrow F\ T' \\
T' \rightarrow *\ F\ T' \\
F \rightarrow (\ E\ ) \mid \textbf{id}
\end{array}
$$

se obtiene mediante la eliminación de la recursividad inmediata por la izquierda de la gramática de expresiones (4.1). El par recursivo por la izquierda de las producciones $E \rightarrow E + T \mid T$ se sustituye mediante $E \rightarrow T\ E'$ y $E' \rightarrow +\ T\ E' \mid \epsilon$. Las nuevas producciones para T y T' se obtienen de manera similar, eliminando la recursividad inmediata por la izquierda. □

La recursividad inmediata por la izquierda puede eliminarse mediante la siguiente técnica, que funciona para cualquier número de producciones A. En primer lugar, se agrupan las producciones de la siguiente manera:

$$A \rightarrow A\alpha_1 \mid A\alpha_2 \mid \cdots \mid A\alpha_m \mid \beta_1 \mid \beta_2 \mid \cdots \mid \beta_n$$

en donde ninguna β_i termina con una A. Después, se sustituyen las producciones A mediante lo siguiente:

$$A \rightarrow \beta_1 A' \mid \beta_2 A' \mid \cdots \mid \beta_n A'$$
$$A' \rightarrow \alpha_1 A' \mid \alpha_2 A' \mid \cdots \mid \alpha_m A' \mid \epsilon$$

El no terminal A genera las mismas cadenas que antes, pero ya no es recursiva por la izquierda. Este procedimiento elimina toda la recursividad por la izquierda de las producciones A y A' (siempre y cuando ninguna α_i sea ϵ), pero no elimina la recursividad por la izquierda que incluye a las derivaciones de dos o más pasos. Por ejemplo, considere la siguiente gramática:

$$\begin{array}{l} S \rightarrow A\ a \mid b \\ A \rightarrow A\ c \mid S\ d \mid \epsilon \end{array} \tag{4.18}$$

El no terminal S es recursiva por la izquierda, ya que $S \Rightarrow Aa \Rightarrow Sda$, pero no es inmediatamente recursiva por la izquierda.

El Algoritmo 4.19, que se muestra a continuación, elimina en forma sistemática la recursividad por la izquierda de una gramática. Se garantiza que funciona si la gramática no tiene ciclos (derivaciones de la forma $A \stackrel{+}{\Rightarrow} A$) o producciones ϵ (producciones de la forma $A \rightarrow \epsilon$). Los ciclos pueden eliminarse en forma sistemática de una gramática, al igual que las producciones ϵ (vea los ejercicios 4.4.6 y 4.4.7).

Algoritmo 4.19: Eliminación de la recursividad por la izquierda.

ENTRADA: La gramática G sin ciclos ni producciones ϵ.

SALIDA: Una gramática equivalente sin recursividad por la izquierda.

MÉTODO: Aplicar el algoritmo de la figura 4.11 a G. Observe que la gramática no recursiva por la izquierda resultante puede tener producciones ϵ. □

```
1)  ordenar los no terminales de cierta forma A1, A2, ..., An.
2)  for ( cada i de 1 a n ) {
3)      for ( cada j de 1 a i − 1 ) {
4)          sustituir cada producción de la forma Ai → Ajγ por las
                producciones Ai → δ1γ | δ2γ | ··· | δkγ, en donde
                Aj → δ1 | δ2 | ··· | δk sean todas producciones Aj actuales
5)      }
6)      eliminar la recursividad inmediata por la izquierda entre las producciones Ai
7)  }
```

Figura 4.11: Algoritmo para eliminar la recursividad por la izquierda de una gramática

El procedimiento en la figura 4.11 funciona de la siguiente manera. En la primera iteración para $i = 1$, el ciclo for externo de las líneas (2) a la (7) elimina cualquier recursividad inmediata por la izquierda entre las producciones A_1. Cualquier producción A_1 restante de la forma $A_1 \rightarrow A_l\alpha$ debe, por lo tanto, tener $l > 1$. Después de la i-1-ésima iteración del ciclo for externo, todas las no terminales A_k, en donde $k < i$, se "limpian"; es decir, cualquier producción $A_k \rightarrow A_l\alpha$ debe tener $l > k$. Como resultado, en la i-ésima iteración, el ciclo interno de las líneas (3) a la (5) eleva en forma progresiva el límite inferior en cualquier producción $A_i \rightarrow A_m\alpha$, hasta tener $m \geq i$.

Después, la eliminación de la recursividad inmediata por la izquierda para las producciones A_i en la línea (6) obliga a que m sea mayor que i.

Ejemplo 4.20: Vamos a aplicar el Algoritmo 4.19 a la gramática (4.18). Técnicamente, no se garantiza que el algoritmo vaya a funcionar debido a la producción ϵ, pero en este caso, la producción $A \rightarrow \epsilon$ resulta ser inofensiva.

Ordenamos los no terminales S, A. No hay recursividad inmediata por la izquierda entre las producciones S, por lo que no ocurre nada durante el ciclo externo para $i = 1$. Para $i = 2$, sustituimos la S en $A \rightarrow S\ d$ para obtener las siguientes producciones A.

$$A \rightarrow A\ c \mid A\ a\ d \mid b\ d \mid \epsilon$$

Al eliminar la recursividad inmediata por la izquierda entre las producciones A produce la siguiente gramática:

$$\begin{array}{l} S \rightarrow A\ a \mid b \\ A \rightarrow b\ d\ A' \mid A' \\ A' \rightarrow c\ A' \mid a\ d\ A' \mid \epsilon \end{array}$$

□

4.3.4 Factorización por la izquierda

La factorización por la izquierda es una transformación gramátical, útil para producir una gramática adecuada para el análisis sintáctico predictivo, o descendente. Cuando la elección entre dos producciones A alternativas no está clara, tal vez podamos rescribir las producciones para diferir la decisión hasta haber visto la suficiente entrada como para poder realizar la elección correcta.

Por ejemplo, si tenemos las siguientes dos producciones:

$$\begin{array}{rcl} \textit{instr} & \rightarrow & \textbf{if}\ \textit{expr}\ \textbf{then}\ \textit{instr}\ \textbf{else}\ \textit{instr} \\ & \mid & \textbf{if}\ \textit{expr}\ \textbf{then}\ \textit{instr} \end{array}$$

al ver la entrada **if**, no podemos saber de inmediato qué producción elegir para expandir *instr*. En general, si $A \rightarrow \alpha\beta_1 \mid \alpha\beta_2$ son dos producciones A, y la entrada empieza con una cadena no vacía derivada de α, no sabemos si debemos expandir A a $\alpha\beta_1$ o a $\alpha\beta_2$. No obstante, podemos diferir la decisión si expandimos A a $\alpha A'$. Así, después de ver la entrada derivada de α, expandimos A' a β_1 o a β_2. Es decir, si se factorizan por la izquierda, las producciones originales se convierten en:

$$\begin{array}{l} A \rightarrow \alpha A' \\ A' \rightarrow \beta_1 \mid \beta_2 \end{array}$$

Algoritmo 4.21: Factorización por la izquierda de una gramática.

ENTRADA: La gramática G.

SALIDA: Una gramática equivalente factorizada por la izquierda.

MÉTODO: Para cada no terminal A, encontrar el prefijo α más largo que sea común para una o más de sus alternativas. Si $\alpha \neq \epsilon$ (es decir, si hay un prefijo común no trivial), se sustituyen todas las producciones A, $A \rightarrow \alpha\beta_1 \mid \alpha\beta_2 \mid \cdots \mid \alpha\beta_n \mid \gamma$, en donde γ representa a todas las alternativas que no empiezan con α, mediante lo siguiente:

$$\begin{array}{l} A \rightarrow \alpha A' \mid \gamma \\ A' \rightarrow \beta_1 \mid \beta_2 \mid \cdots \mid \beta_n \end{array}$$

Aquí, A' es un no terminal nuevo. Se aplica esta transformación en forma repetida hasta que no haya dos alternativas para un no terminal que tengan un prefijo común. □

Ejemplo 4.22: La siguiente gramática abstrae el problema del "else colgante":

$$\begin{array}{l} S \rightarrow i\ E\ t\ S \mid i\ E\ t\ S\ e\ S \mid a \\ E \rightarrow b \end{array} \tag{4.23}$$

Aquí, i, t y e representan a **if**, **then** y **else**; E y S representan "expresión condicional" e "instrucción". Si se factoriza a la izquierda, esta gramática se convierte en:

$$\begin{array}{l} S \rightarrow i\ E\ t\ S\ S' \mid a \\ S' \rightarrow e\ S \mid \epsilon \\ E \rightarrow b \end{array} \tag{4.24}$$

Así, podemos expandir S a $iEtSS'$ con la entrada i, y esperar hasta que se haya visto $iEtS$ para decidir si se va a expandir S' a eS o a ϵ. Desde luego que estas gramáticas son ambiguas, y con la entrada e no quedará claro qué alternativa debe elegirse para S'. El ejemplo 4.33 habla sobre cómo salir de este dilema. □

4.3.5 Construcciones de lenguajes que no son libres de contexto

Algunas construcciones sintácticas que se encuentran en los lenguajes de programación ordinarios no pueden especificarse sólo mediante el uso de gramáticas. Aquí consideraremos dos de estas construcciones, usando lenguajes abstractos simples para ilustrar las dificultades.

Ejemplo 4.25: El lenguaje en este ejemplo abstrae el problema de comprobar que se declaren los identificadores antes de poder usarlos en un programa. El lenguaje consiste en cadenas de la forma wcw, en donde la primera w representa la declaración de un identificador w, c representa un fragmento intermedio del programa, y la segunda w representa el uso del identificador.

El lenguaje abstracto es $L_1 = \{wcw \mid w \text{ está en } (\mathbf{a}|\mathbf{b})^*\}$. L_1 consiste en todas las palabras compuestas de una cadena repetida de as y bs separadas por c, como $aabcaab$. Aunque no lo vamos a demostrar aquí, la característica de no ser libre de contexto de L_1 implica directamente que los lenguajes de programación como C y Java no sean libres de contexto, los cuales requieren la declaración de los identificadores antes de usarlos, además de permitir identificadores de longitud arbitraria.

Por esta razón, una gramática para C o Java no hace diferencias entre los identificadores que son cadenas distintas de caracteres. En vez de ello, todos los identificadores se representan

mediante un token como **id** en la gramática. En un compilador para dicho lenguaje, la fase de análisis semántico comprueba que los identificadores se declaren antes de usarse. □

Ejemplo 4.26: El lenguaje, que no es independiente del contexto en este ejemplo, abstrae el problema de comprobar que el número de parámetros formales en la declaración de una función coincida con el número de parámetros actuales en un uso de la función. El lenguaje consiste en cadenas de la forma $a^n b^m c^n d^m$. (Recuerde que a^n significa a escrita n veces). Aquí, a^n y b^m podrían representar las listas de parámetros formales de dos funciones declaradas para tener n y m argumentos, respectivamente, mientras que c^n y d^m representan las listas de los parámetros actuales en las llamadas a estas dos funciones.

El lenguaje abstracto es $L_2 = \{a^n b^m c^n d^m \mid n \geq 1 \text{ y } m \geq 1\}$. Es decir, L_2 consiste de cadenas en el lenguaje generado por la expresión regular $\mathbf{a}^*\mathbf{b}^*\mathbf{c}^*\mathbf{d}^*$, de tal forma que el número de as y cs y de bs y ds sea igual. Este lenguaje no es independiente del contexto.

De nuevo, la sintaxis común de las declaraciones de las funciones y los usos no se responsabiliza de contar el número de parámetros. Por ejemplo, la llamada a una función en un lenguaje similar a C podría especificarse de la siguiente manera:

$$\begin{array}{rcl} \mathit{instr} & \rightarrow & \mathbf{id}\ (\mathit{lista\_expr}) \\ \mathit{lista\_expr} & \rightarrow & \mathit{lista\_expr},\ \mathit{expr} \\ & \mid & \mathit{expr} \end{array}$$

con producciones adecuadas para *expr*. La comprobación de que el número de parámetros en una llamada sea correcto se realiza por lo general durante la fase del análisis semántico. □

4.3.6 Ejercicios para la sección 4.3

Ejercicio 4.3.1: La siguiente gramática es para expresiones regulares sobre los símbolos a y b solamente, usando + en vez de | para la unión, con lo cual se evita el conflicto con el uso de la barra vertical como un meta símbolo en las gramáticas:

$$\begin{array}{lcl} \mathit{rexpr} & \rightarrow & \mathit{rexpr} + \mathit{rterm} \mid \mathit{rterm} \\ \mathit{rterm} & \rightarrow & \mathit{rterm}\ \mathit{rfactor} \mid \mathit{rfactor} \\ \mathit{rfactor} & \rightarrow & \mathit{rfactor} * \mid \mathit{rprimario} \\ \mathit{rprimario} & \rightarrow & \mathbf{a} \mid \mathbf{b} \end{array}$$

a) Factorice esta gramática por la izquierda.

b) ¿La factorización por la izquierda hace a la gramática adecuada para el análisis sintáctico descendente?

c) Además de la factorización por la izquierda, elimine la recursividad por la izquierda de la gramática original.

d) ¿La gramática resultante es adecuada para el análisis sintáctico descendente?

Ejercicio 4.3.2: Repita el ejercicio 4.3.1 con las siguientes gramáticas:

a) La gramática del ejercicio 4.2.1.

b) La gramática del ejercicio 4.2.2(a).

c) La gramática del ejercicio 4.2.2(c).

d) La gramática del ejercicio 4.2.2(e).

e) La gramática del ejercicio 4.2.2(g).

! **Ejercicio 4.3.3:** Se propone la siguiente gramática para eliminar la "ambigüedad del else colgante", descrita en la sección 4.3.2:

$$\begin{array}{lcl} \textit{instr} & \rightarrow & \textbf{if } \textit{expr} \textbf{ then } \textit{instr} \\ & | & \textit{instrRelacionada} \\ \textit{instrRelacionada} & \rightarrow & \textbf{if } \textit{expr} \textbf{ then } \textit{instrRelacionada} \textbf{ else } \textit{instr} \\ & | & \textbf{otra} \end{array}$$

Muestre que esta gramática sigue siendo ambigua.

4.4 Análisis sintáctico descendente

El análisis sintáctico descendente puede verse como el problema de construir un árbol de análisis sintáctico para la cadena de entrada, partiendo desde la raíz y creando los nodos del árbol de análisis sintáctico en preorden (profundidad primero, como vimos en la sección 2.3.4). De manera equivalente, podemos considerar el análisis sintáctico descendente como la búsqueda de una derivación por la izquierda para una cadena de entrada.

Ejemplo 4.27: La secuencia de árboles de análisis sintáctico en la figura 4.12 para la entrada **id+id*id** es un análisis sintáctico descendente, de acuerdo con la gramática (4.2), que repetimos a continuación:

$$\begin{array}{lcl} E & \rightarrow & T\ E' \\ E' & \rightarrow & +\ T\ E' \ \mid\ \epsilon \\ T & \rightarrow & F\ T' \\ T' & \rightarrow & *\ F\ T' \ \mid\ \epsilon \\ F & \rightarrow & (\ E\ ) \ \mid\ \textbf{id} \end{array} \tag{4.28}$$

Esta secuencia de árboles corresponde a una derivación por la izquierda de la entrada. □

En cada paso de un análisis sintáctico descendente, el problema clave es el de determinar la producción que debe aplicarse para un no terminal, por decir A. Una vez que se elige una producción A, el resto del proceso de análisis sintáctico consiste en "relacionar" los símbolos terminales en el cuerpo de la producción con la cadena de entrada.

Esta sección empieza con una forma general del análisis sintáctico descendente, conocida como análisis sintáctico de descenso recursivo, la cual puede requerir de un rastreo hacia atrás para encontrar la producción A correcta que debe aplicarse. La sección 2.4.2 introdujo el análisis sintáctico predictivo, un caso especial de análisis sintáctico de descenso recursivo, en donde no se requiere un rastreo hacia atrás. El análisis sintáctico predictivo elige la producción A correcta mediante un análisis por adelantado de la entrada, en donde se ve un número fijo de símbolos adelantados; por lo general, sólo necesitamos ver un símbolo por adelantado (es decir, el siguiente símbolo de entrada).

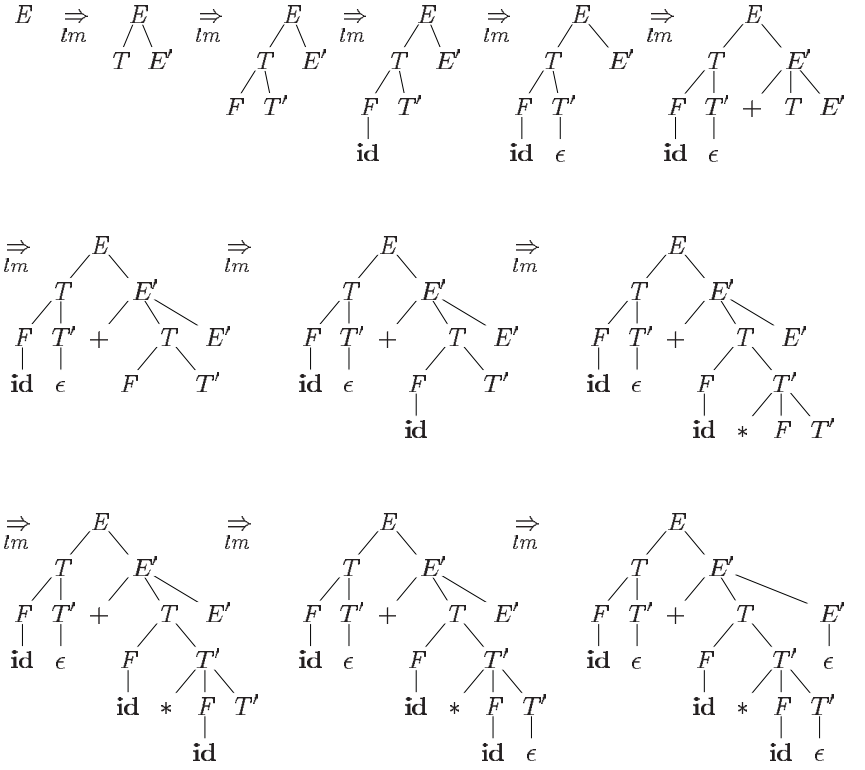

Figura 4.12: Análisis sintáctico descendente para **id**+**id**∗**id**

Por ejemplo, considere el análisis sintáctico descendente en la figura 4.12, en la cual se construye un árbol con dos nodos etiquetados como E'. En el primer nodo E' (en preorden), se elige la producción $E' \rightarrow +TE'$; en el segundo nodo E', se elige la producción $E' \rightarrow \epsilon$. Un analizador sintáctico predictivo puede elegir una de las producciones E' mediante el análisis del siguiente símbolo de entrada.

A la clase de gramáticas para las cuales podemos construir analizadores sintácticos predictivos que analicen k símbolos por adelantado en la entrada, se le conoce algunas veces como la clase $LL(k)$. En la sección 4.4.3 hablaremos sobre la clase LL(1), pero presentaremos antes ciertos cálculos, llamados PRIMERO y SIGUIENTE, en la sección 4.4.2. A partir de los conjuntos PRIMERO y SIGUIENTE para una gramática, construiremos "tablas de análisis sintáctico predictivo", las cuales hacen explícita la elección de la producción durante el análisis sintáctico descendente. Estos conjuntos también son útiles durante el análisis sintáctico ascendente.

En la sección 4.4.4 proporcionaremos un algoritmo de análisis sintáctico no recursivo que mantiene una pila en forma explícita, en vez de hacerlo en forma implícita mediante llamadas recursivas. Por último, en la sección 4.4.5 hablaremos sobre la recuperación de errores durante el análisis sintáctico descendente.

4.4.1 Análisis sintáctico de descenso recursivo

```
     void A() {
1)       Elegir una producción A, A → X1X2 ··· Xk;
2)       for ( i = 1 a k ) {
3)           if ( Xi es un no terminal )
4)               llamar al procedimiento Xi();
5)           else if ( Xi es igual al símbolo de entrada actual a )
6)               avanzar la entrada hasta el siguiente símbolo;
7)           else /* ha ocurrido un error */;
         }
     }
```

Figura 4.13: Un procedimiento ordinario para un no terminal en un analizador sintáctico descendente

Un programa de análisis sintáctico de descenso recursivo consiste en un conjunto de procedimientos, uno para cada no terminal. La ejecución empieza con el procedimiento para el símbolo inicial, que se detiene y anuncia que tuvo éxito si el cuerpo de su procedimiento explora toda la cadena completa de entrada. En la figura 4.13 aparece el seudocódigo para un no terminal común. Observe que este seudocódigo es no determinista, ya que empieza eligiendo la producción A que debe aplicar de una forma no especificada.

El descenso recursivo general puede requerir de un rastreo hacia atrás; es decir, tal vez requiera exploraciones repetidas sobre la entrada. Sin embargo, raras veces se necesita el rastreo hacia atrás para analizar las construcciones de un lenguaje de programación, por lo que los analizadores sintácticos con éste no se ven con frecuencia. Incluso para situaciones como el análisis sintáctico de un lenguaje natural, el rastreo hacia atrás no es muy eficiente, por lo cual se prefieren métodos tabulares como el algoritmo de programación dinámico del ejercicio 4.4.9, o el método de Earley (vea las notas bibliográficas).

Para permitir el rastreo hacia atrás, hay que modificar el código de la figura 4.13. En primer lugar, no podemos elegir una producción A única en la línea (1), por lo que debemos probar cada una de las diversas producciones en cierto orden. Después, el fallo en la línea (7) no es definitivo, sino que sólo sugiere que necesitamos regresar a la línea (1) y probar otra producción A. Sólo si no hay más producciones A para probar es cuando declaramos que se ha encontrado un error en la entrada. Para poder probar otra producción A, debemos restablecer el apuntador de entrada a la posición en la que se encontraba cuando llegamos por primera vez a la línea (1). Es decir, se requiere una variable local para almacenar este apuntador de entrada, para un uso futuro.

Ejemplo 4.29: Considere la siguiente gramática:

$$\begin{array}{lcl} S & \rightarrow & c\ A\ d \\ A & \rightarrow & a\ b \mid a \end{array}$$

Para construir un árbol de análisis sintáctico descendente para la cadena de entrada $w = cad$, empezamos con un árbol que consiste en un solo nodo etiquetado como S, y el apuntador de entrada apunta a c, el primer símbolo de w. S sólo tiene una producción, por lo que la utilizamos

para expandir S y obtener el árbol de la figura 4.14(a). La hoja por la izquierda, etiquetada como c, coincide con el primer símbolo de la entrada w, por lo que avanzamos el apuntador de entrada hasta a, el segundo símbolo de w, y consideramos la siguiente hoja, etiquetada como A.

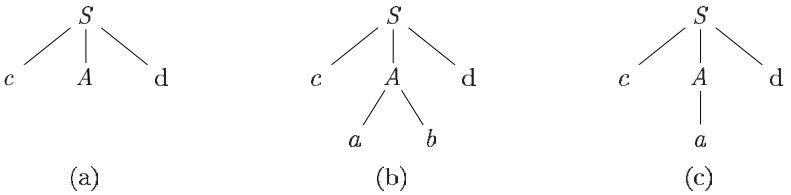

Figura 4.14: Los pasos de un análisis sintáctico descendente

Ahora expandimos A mediante la primera alternativa $A \rightarrow a\ b$ para obtener el árbol de la figura 4.14(b). Tenemos una coincidencia para el segundo símbolo de entrada a, por lo que avanzamos el apuntador de entrada hasta d, el tercer símbolo de entrada, y comparamos a d con la siguiente hoja, etiquetada con b. Como b no coincide con d, reportamos un error y regresamos a A para ver si hay otra alternativa para A que no hayamos probado y que pueda producir una coincidencia.

Al regresar a A, debemos restablecer el apuntador de entrada a la posición 2, la posición que tenía cuando llegamos por primera vez a A, lo cual significa que el procedimiento para A debe almacenar el apuntador de entrada en una variable local.

La segunda alternativa para A produce el árbol de la figura 4.14(c). La hoja a coincide con el segundo símbolo de w y la hoja d coincide con el tercer símbolo. Como hemos producido un árbol de análisis sintáctico para w, nos detenemos y anunciamos que se completó el análisis sintáctico con éxito. □

Una gramática recursiva por la izquierda puede hacer que un analizador sintáctico de descenso recursivo, incluso uno con rastreo hacia atrás, entre en un ciclo infinito. Es decir, al tratar de expandir una no terminal A, podríamos en un momento dado encontrarnos tratando otra vez de expandir a A, sin haber consumido ningún símbolo de la entrada.

4.4.2 PRIMERO y SIGUIENTE

La construcción de los analizadores sintácticos descendentes y ascendentes es auxiliada por dos funciones, PRIMERO y SIGUIENTE, asociadas con la gramática G. Durante el análisis sintáctico descendente, PRIMERO y SIGUIENTE nos permiten elegir la producción que vamos a aplicar, con base en el siguiente símbolo de entrada. Durante la recuperación de errores en modo de pánico, los conjuntos de tokens que produce SIGUIENTE pueden usarse como tokens de sincronización.

Definimos a *PRIMERO*(α), en donde α es cualquier cadena de símbolos gramaticales, como el conjunto de terminales que empiezan las cadenas derivadas a partir de α. Si $\alpha \overset{*}{\Rightarrow} \epsilon$, entonces ϵ también se encuentra en PRIMERO(α). Por ejemplo, en la figura 4.15, $A \overset{*}{\underset{lm}{\Rightarrow}} c\gamma$, por lo que c está en PRIMERO(A).

Para una vista previa de cómo usar PRIMERO durante el análisis sintáctico predictivo, considere dos producciones A, $A \rightarrow \alpha \mid \beta$, en donde PRIMERO$(\alpha)$ y PRIMERO(β) son conjuntos

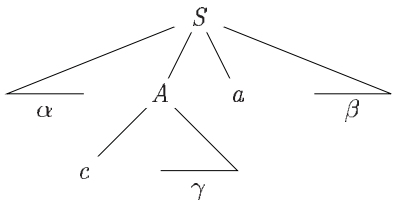

Figura 4.15: El terminal c está en PRIMERO(A) y a está en SIGUIENTE(A)

separados. Entonces, podemos elegir una de estas producciones A si analizamos el siguiente símbolo de entrada a, ya que a puede estar a lo más en PRIMERO(α) o en PRIMERO(β), pero no en ambos. Por ejemplo, si a está en PRIMERO(β), elegimos la producción $A \rightarrow \beta$. Exploraremos esta idea en la sección 4.4.3, cuando definamos las gramáticas LL(1).

Definimos a *SIGUIENTE*(A), para el no terminal A, como el conjunto de terminales a que pueden aparecer de inmediato a la derecha de A en cierta forma de frase; es decir, el conjunto de terminales A de tal forma que exista una derivación de la forma $S \stackrel{*}{\Rightarrow} \alpha A a \beta$, para algunas α y β, como en la figura 4.15. Observe que pudieron haber aparecido símbolos entre A y a, en algún momento durante la derivación, pero si es así, derivaron a ϵ y desaparecieron. Además, si A puede ser el símbolo por la derecha en cierta forma de frase, entonces \$ está en SIGUIENTE(A); recuerde que \$ es un símbolo "delimitador" especial, el cual se supone que no es un símbolo de ninguna gramática.

Para calcular PRIMERO(X) para todos los símbolos gramaticales X, aplicamos las siguientes reglas hasta que no pueden agregarse más terminales o ϵ a ningún conjunto PRIMERO.

1. Si X es un terminal, entonces PRIMERO(X) = $\{X\}$.

2. Si X es un no terminal y $X \rightarrow Y_1 Y_2 \cdots Y_k$ es una producción para cierta $k \geq 1$, entonces se coloca a en PRIMERO(X) si para cierta i, a está en PRIMERO(Y_i), y ϵ está en todas las funciones PRIMERO(Y_1), ..., PRIMERO(Y_{i-1}); es decir, $Y_1 \cdots Y_{i-1} \stackrel{*}{\Rightarrow} \epsilon$. Si ϵ está en PRIMERO(Y_j) para todas las $j = 1, 2, \ldots, k$, entonces se agrega ϵ a PRIMERO(X). Por ejemplo, todo lo que hay en PRIMERO(Y_1) se encuentra sin duda en PRIMERO(X). Si Y_1 no deriva a ϵ, entonces no agregamos nada más a PRIMERO(X), pero si $Y_1 \stackrel{*}{\Rightarrow} \epsilon$, entonces agregamos PRIMERO(Y_2), y así sucesivamente.

3. Si $X \rightarrow \epsilon$ es una producción, entonces se agrega ϵ a PRIMERO(X).

Ahora, podemos calcular PRIMERO para cualquier cadena $X_1 X_2 \cdots X_n$ de la siguiente manera. Se agregan a PRIMERO($X_1 X_2 \cdots X_n$) todos los símbolos que no sean ϵ de PRIMERO(X_1). También se agregan los símbolos que no sean ϵ de PRIMERO(X_2), si ϵ está en PRIMERO(X_1); los símbolos que no sean ϵ de PRIMERO(X_3), si ϵ está en PRIMERO(X_1) y PRIMERO(X_2); y así sucesivamente. Por último, se agrega ϵ a PRIMERO($X_1 X_2 \cdots X_n$) si, para todas las i, ϵ se encuentra en PRIMERO(X_i).

Para calcular SIGUIENTE(A) para todas las no terminales A, se aplican las siguientes reglas hasta que no pueda agregarse nada a cualquier conjunto SIGUIENTE.

1. Colocar \$ en SIGUIENTE(S), en donde S es el símbolo inicial y \$ es el delimitador derecho de la entrada.

2. Si hay una producción $A \rightarrow \alpha B \beta$, entonces todo lo que hay en PRIMERO(β) excepto ϵ está en SIGUIENTE(B).

3. Si hay una producción $A \rightarrow \alpha B$, o una producción $A \rightarrow \alpha B \beta$, en donde PRIMERO($\beta$) contiene a ϵ, entonces todo lo que hay en SIGUIENTE(A) está en SIGUIENTE(B).

Ejemplo 4.30: Considere de nuevo la gramática no recursiva por la izquierda (4.28). Entonces:

1. PRIMERO(F) = PRIMERO(T) = PRIMERO(E) = {(, **id**}. Para ver por qué, observe que las dos producciones para F tienen producciones que empiezan con estos dos símbolos terminales, **id** y el paréntesis izquierdo. T sólo tiene una producción y empieza con F. Como F no deriva a ϵ, PRIMERO(T) debe ser igual que PRIMERO(F). El mismo argumento se cumple con PRIMERO(E).

2. PRIMERO(E') = {+, ϵ}. La razón es que una de las dos producciones para E' tiene un cuerpo que empieza con el terminal +, y la otra es ϵ. Cada vez que un no terminal deriva a ϵ, colocamos a ϵ en PRIMERO para ese no terminal.

3. PRIMERO(T') = {$*$, ϵ}. El razonamiento es análogo al de PRIMERO(E').

4. SIGUIENTE(E) = SIGUIENTE(E') = {), \$}. Como E es el símbolo inicial, SIGUIENTE(E) debe contener \$. El cuerpo de la producción (E) explica por qué el paréntesis derecho está en SIGUIENTE(E). Para E', observe que esta no terminal sólo aparece en los extremos de los cuerpos de las producciones E. Por ende, SIGUIENTE(E') debe ser el mismo que SIGUIENTE(E).

5. SIGUIENTE(T) = SIGUIENTE(T') = {+,), \$}. Observe que T aparece en las producciones sólo seguido por E'. Por lo tanto, todo lo que esté en PRIMERO(E'), excepto ϵ, debe estar en SIGUIENTE(T); eso explica el símbolo +. No obstante, como PRIMERO(E') contiene a ϵ (es decir, $E' \stackrel{*}{\Rightarrow} \epsilon$), y E' es la cadena completa que va después de T en los cuerpos de las producciones E, todo lo que hay en SIGUIENTE(E) también debe estar en SIGUIENTE(T). Eso explica los símbolos \$ y el paréntesis derecho. En cuanto a T', como aparece sólo en los extremos de las producciones T, debe ser que SIGUIENTE(T') = SIGUIENTE(T).

6. SIGUIENTE(F) = {+, $*$,), \$}. El razonamiento es análogo al de T en el punto (5).

□

4.4.3 Gramáticas LL(1)

Los analizadores sintácticos predictivos, es decir, los analizadores sintácticos de descenso recursivo que no necesitan rastreo hacia atrás, pueden construirse para una clase de gramáticas llamadas LL(1). La primera "L" en LL(1) es para explorar la entrada de izquierda a derecha (por left en inglés), la segunda "L" para producir una derivación por la izquierda, y el "1" para usar un símbolo de entrada de anticipación en cada paso, para tomar las decisiones de acción del análisis sintáctico.

Diagramas de transición para analizadores sintácticos predictivos

Los diagramas de transición son útiles para visualizar los analizadores sintácticos predictivos. Por ejemplo, los diagramas de transición para las no terminales E y E' de la gramática (4.28) aparecen en la figura 4.16(a). Para construir el diagrama de transición a partir de una gramática, primero hay que eliminar la recursividad por la izquierda y después factorizar la gramática por la izquierda. Entonces, para cada no terminal A,

1. Se crea un estado inicial y un estado final (retorno).
2. Para cada producción $A \rightarrow X_1X_2 \cdots X_k$, se crea una ruta desde el estado inicial hasta el estado final, con los flancos etiquetados como X_1, X_2, ..., X_k. Si $A \rightarrow \epsilon$, la ruta es una línea que se etiqueta como ϵ.

Los diagramas de transición para los analizadores sintácticos predictivos difieren de los diagramas para los analizadores léxicos. Los analizadores sintácticos tienen un diagrama para cada no terminal. Las etiquetas de las líneas pueden ser tokens o no terminales. Una transición sobre un token (terminal) significa que tomamos esa transición si ese token es el siguiente símbolo de entrada. Una transición sobre un no terminal A es una llamada al procedimiento para A.

Con una gramática LL(1), la ambigüedad acera de si se debe tomar o no una línea ϵ puede resolverse si hacemos que las transiciones ϵ sean la opción predeterminada.

Los diagramas de transición pueden simplificarse, siempre y cuando se preserve la secuencia de símbolos gramaticales a lo largo de las rutas. También podemos sustituir el diagrama para un no terminal A, en vez de una línea etiquetada como A. Los diagramas en las figuras 4.16(a) y (b) son equivalentes: si trazamos rutas de E a un estado de aceptación y lo sustituimos por E', entonces, en ambos conjuntos de diagramas, los símbolos gramaticales a lo largo de las rutas forman cadenas del tipo $T + T + \cdots + T$. El diagrama en (b) puede obtenerse a partir de (a) mediante transformaciones semejantes a las de la sección 2.5.4, en donde utilizamos la eliminación de recursividad de la parte final y la sustitución de los cuerpos de los procedimientos para optimizar el procedimiento en un no terminal.

La clase de gramáticas LL(1) es lo bastante robusta como para cubrir la mayoría de las construcciones de programación, aunque hay que tener cuidado al escribir una gramática adecuada para el lenguaje fuente. Por ejemplo, ninguna gramática recursiva por la izquierda o ambigua puede ser LL(1).

Una gramática G es LL(1) si, y sólo si cada vez que $A \rightarrow \alpha \mid \beta$ son dos producciones distintas de G, se aplican las siguientes condiciones:

1. Para el no terminal a, tanto α como β derivan cadenas que empiecen con a.
2. A lo más, sólo α o β puede derivar la cadena vacía.
3. Si $\beta \stackrel{*}{\Rightarrow} \epsilon$, entonces α no deriva a ninguna cadena que empiece con una terminal en SIGUIENTE(A). De igual forma, si $\alpha \stackrel{*}{\Rightarrow} \epsilon$, entonces β no deriva a ninguna cadena que empiece con una terminal en SIGUIENTE(A).

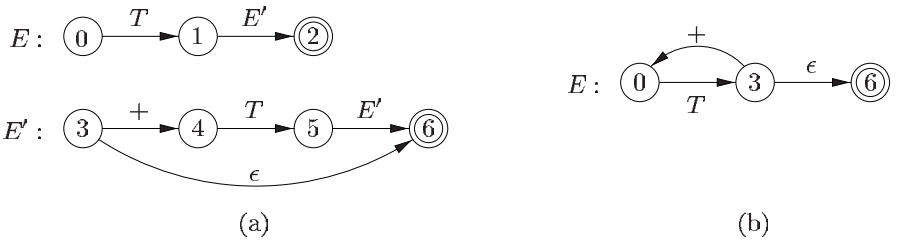

Figura 4.16: Diagramas de transición para los no terminales E y E' de la gramática 4.28

Las primeras dos condiciones son equivalentes para la instrucción que establece que PRIMERO(α) y PRIMERO(β) son conjuntos separados. La tercera condición equivale a decir que si ϵ está en PRIMERO(β), entonces PRIMERO(α) y SIGUIENTE(A) son conjuntos separados, y de igual forma si ϵ está en PRIMERO(α).

Pueden construirse analizadores sintácticos predictivos para las gramáticas LL(1), ya que puede seleccionarse la producción apropiada a aplicar para una no terminal con sólo analizar el símbolo de entrada actual. Los constructores del flujo de control, con sus palabras clave distintivas, por lo general, cumplen con las restricciones de LL(1). Por ejemplo, si tenemos las siguientes producciones:

$$\begin{array}{rcl} \textit{instr} & \rightarrow & \textbf{if} \ (\ \textit{expr}\ )\ \textit{instr}\ \textbf{else}\ \textit{instr} \\ & | & \textbf{while}\ (\ \textit{expr}\ )\ \ \textit{instr} \\ & | & \{\ \textit{lista\_instr}\ \} \end{array}$$

entonces las palabras clave **if**, **while** y el símbolo { nos indican qué alternativa es la única que quizá podría tener éxito, si vamos a buscar una instrucción.

El siguiente algoritmo recolecta la información de los conjuntos PRIMERO y SIGUIENTE en una tabla de análisis predictivo $M[A, a]$, un arreglo bidimensional, en donde A es un no terminal y a es un terminal o el símbolo \$, el marcador de fin de la entrada. El algoritmo se basa en la siguiente idea: se elige la producción $A \rightarrow \alpha$ si el siguiente símbolo de entrada a se encuentra en PRIMERO(α). La única complicación ocurre cuando $\alpha = \epsilon$, o en forma más general, $\alpha \stackrel{*}{\Rightarrow} \epsilon$. En este caso, debemos elegir de nuevo $A \rightarrow \alpha$ si el símbolo de entrada actual se encuentra en SIGUIENTE(A), o si hemos llegado al \$ en la entrada y \$ se encuentra en SIGUIENTE(A).

Algoritmo 4.31: Construcción de una tabla de análisis sintáctico predictivo.

ENTRADA: La gramática G.

SALIDA: La tabla de análisis sintáctico M.

MÉTODO: Para cada producción $A \rightarrow \alpha$ de la gramática, hacer lo siguiente:

1. Para cada terminal a en PRIMERO(A), agregar $A \rightarrow \alpha$ a $M[A, a]$.
2. Si ϵ está en PRIMERO(α), entonces para cada terminal b en SIGUIENTE(A), se agrega $A \rightarrow \alpha$ a $M[A, b]$. Si ϵ está en PRIMERO(α) y \$ se encuentra en SIGUIENTE(A), se agrega $A \rightarrow \alpha$ a $M[A, \$]$ también.

Si después de realizar lo anterior, no hay producción en $M[A, a]$, entonces se establece $M[A, a]$ a **error** (que, por lo general, representamos mediante una entrada vacía en la tabla). □

Ejemplo 4.32: Para la gramática de expresiones (4.28), el Algoritmo 4.31 produce la tabla de análisis sintáctico en la figura 4.17. Los espacios en blanco son entradas de error; los espacios que no están en blanco indican una producción con la cual se expande un no terminal.

No terminal	Símbolo de entrada					
	id	+	*	(	)	$
E	$E \to TE'$			$E \to TE'$		
E'		$E' \to +TE'$			$E' \to \epsilon$	$E' \to \epsilon$
T	$T \to FT'$			$T \to FT'$		
T'		$T' \to \epsilon$	$T' \to *FT'$		$T' \to \epsilon$	$T' \to \epsilon$
F	$F \to$ **id**			$F \to (E)$		

Figura 4.17: Tabla de análisis sintáctico M para el ejemplo 4.32

Considere la producción $E \to TE'$. Como

$$\text{PRIMERO}(TE') = \text{PRIMERO}(T) = \{(, \textbf{id}\}$$

esta producción se agrega a $M[E, (]$ y $M[E, \textbf{id}]$. La producción $E' \to +TE'$ se agrega a $M[E', +]$, ya que $\text{PRIMERO}(+TE') = \{+\}$. Como $\text{SIGUIENTE}(E') = \{), \$\}$, la producción $E' \to \epsilon$ se agrega a $M[E',)]$ y a $M[E', \$]$. □

El algoritmo 4.31 puede aplicarse a cualquier gramática G para producir una tabla de análisis sintáctico M. Para cada gramática LL(1), cada entrada en la tabla de análisis sintáctico identifica en forma única a una producción, o indica un error. Sin embargo, para algunas gramáticas, M puede tener algunas entradas que tengan múltiples definiciones. Por ejemplo, si G es recursiva por la izquierda o ambigua, entonces M tendrá por lo menos una entrada con múltiples definiciones. Aunque la eliminación de la recursividad por la izquierda y la factorización por la izquierda son fáciles de realizar, hay algunas gramáticas para las cuales ningún tipo de alteración producirá una gramática LL(1).

El lenguaje en el siguiente ejemplo no tiene una gramática LL(1).

Ejemplo 4.33: La siguiente gramática, que abstrae el problema del else colgante, se repite aquí del ejemplo 4.22:

$$\begin{array}{lcl} S & \to & iEtSS' \mid a \\ S' & \to & eS \mid \epsilon \\ E & \to & b \end{array}$$

La tabla de análisis sintáctico para esta gramática aparece en la figura 4.18. La entrada para $M[S', e]$ contiene tanto a $S' \to eS$ como a $S' \to \epsilon$.

La gramática es ambigua y la ambigüedad se manifiesta mediante una elección de qué producción usar cuando se ve una e (**else**). Podemos resolver esta ambigüedad eligiendo $S' \to eS$.

No terminal	Símbolo de entrada					
	a	b	e	i	t	\$
S	$S \to a$			$S \to iEtSS'$		
S'			$S' \to \epsilon$ $S' \to eS$			$S' \to \epsilon$
E		$E \to b$				

Figura 4.18: Tabla de análisis sintáctico M para el ejemplo 4.33

Esta elección corresponde a la asociación de un **else** con el **then** anterior más cercano. Observe que la elección $S' \to \epsilon$ evitaría que e se metiera en la pila o se eliminara de la entrada, y eso definitivamente está mal. □

4.4.4 Análisis sintáctico predictivo no recursivo

Podemos construir un analizador sintáctico predictivo no recursivo mediante el mantenimiento explícito de una pila, en vez de hacerlo mediante llamadas recursivas implícitas. El analizador sintáctico imita una derivación por la izquierda. Si w es la entrada que se ha relacionado hasta ahora, entonces la pila contiene una secuencia de símbolos gramaticales α de tal forma que:

$$S \underset{lm}{\overset{*}{\Rightarrow}} w\alpha$$

El analizador sintáctico controlado por una tabla, que se muestra en la figura 4.19, tiene un búfer de entrada, una pila que contiene una secuencia de símbolos gramaticales, una tabla de análisis sintáctico construida por el Algoritmo 4.31, y un flujo de salida. El búfer de entrada contiene la cadena que se va a analizar, seguida por el marcador final \$. Reutilizamos el símbolo \$ para marcar la parte inferior de la pila, que al principio contiene el símbolo inicial de la gramática encima de \$.

El analizador sintáctico se controla mediante un programa que considera a X, el símbolo en la parte superior de la pila, y a a, el símbolo de entrada actual. Si X es un no terminal, el analizador sintáctico elige una producción X mediante una consulta a la entrada $M[X, a]$ de la tabla de análisis sintáctico M (aquí podría ejecutarse código adicional; por ejemplo, el código para construir un nodo en un árbol de análisis sintáctico). En cualquier otro caso, verifica si hay una coincidencia entre el terminal X y el símbolo de entrada actual a.

El comportamiento del analizador sintáctico puede describirse en términos de sus *configuraciones*, que proporcionan el contenido de la pila y el resto de la entrada. El siguiente algoritmo describe la forma en que se manipulan las configuraciones.

Algoritmo 4.34: Análisis sintáctico predictivo, controlado por una tabla.

ENTRADA: Una cadena w y una tabla de análisis sintáctico M para la gramática G.

SALIDA: Si w está en $L(G)$, una derivación por la izquierda de w; en caso contrario, una indicación de error.

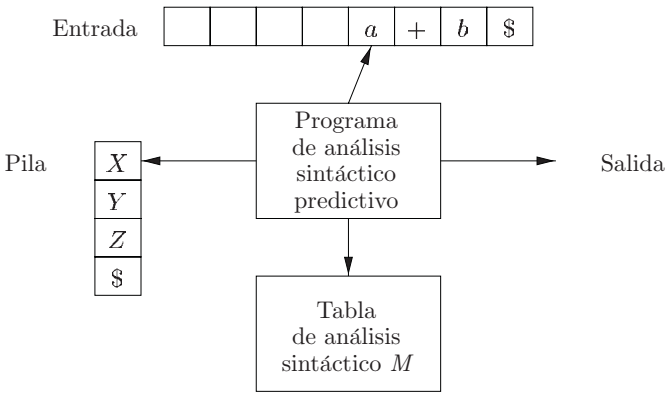

Figura 4.19: Modelo de un analizador sintáctico predictivo, controlado por una tabla

MÉTODO: Al principio, el analizador sintáctico se encuentra en una configuración con $w\$$ en el búfer de entrada, y el símbolo inicial S de G en la parte superior de la pila, por encima de \$. El programa en la figura 4.20 utiliza la tabla de análisis sintáctico predictivo M para producir un análisis sintáctico predictivo para la entrada. □

establecer *ip* para que apunte al primer símbolo de w;
establecer X con el símbolo de la parte superior de la pila;
while ($X \neq \$$) { /* la pila no está vacía */
    if (X es a) sacar de la pila y avanzar *ip*;
    else if (X es un terminal) *error*();
    else if ($M[X, a]$ es una entrada de error) *error*();
    else if ($M[X, a] = X \rightarrow Y_1 Y_2 \cdots Y_k$) {
        enviar de salida la producción $X \rightarrow Y_1 Y_2 \cdots Y_k$;
        sacar de la pila;
        meter Y_k, Y_{k-1}, …, Y_1 en la pila, con Y_1 en la parte superior;
    }
    establecer X con el símbolo de la parte superior de la pila;
}

Figura 4.20: Algoritmo de análisis sintáctico predictivo

Ejemplo 4.35: Considere la gramática (4.28); se muestra la tabla de análisis sintáctico en la figura 4.17. Con la entrada **id** + **id** ∗ **id**, el analizador predictivo sin recursividad del Algoritmo 4.34 realiza la secuencia de movimientos en la figura 4.21. Estos movimientos corresponden a una derivación por la izquierda (vea la figura 4.12 para la derivación completa):

$$E \underset{lm}{\Rightarrow} TE' \underset{lm}{\Rightarrow} FT'E' \underset{lm}{\Rightarrow} \mathbf{id}\ T'E' \underset{lm}{\Rightarrow} \mathbf{id}\ E' \underset{lm}{\Rightarrow} \mathbf{id} + TE' \underset{lm}{\Rightarrow} \cdots$$

COINCIDENCIA	PILA	ENTRADA	ACCIÓN
	$E\$$	$\mathbf{id} + \mathbf{id} * \mathbf{id}\$$	
	$TE'\$$	$\mathbf{id} + \mathbf{id} * \mathbf{id}\$$	emitir $E \to TE'$
	$FT'E'\$$	$\mathbf{id} + \mathbf{id} * \mathbf{id}\$$	emitir $T \to FT'$
	$\mathbf{id}\ T'E'\$$	$\mathbf{id} + \mathbf{id} * \mathbf{id}\$$	emitir $F \to \mathbf{id}$
$\mathbf{id}$	$T'E'\$$	$+\ \mathbf{id} * \mathbf{id}\$$	relacionar $\mathbf{id}$
$\mathbf{id}$	$E'\$$	$+\ \mathbf{id} * \mathbf{id}\$$	emitir $T' \to \epsilon$
$\mathbf{id}$	$+\ TE'\$$	$+\ \mathbf{id} * \mathbf{id}\$$	emitir $E' \to + TE'$
$\mathbf{id} +$	$TE'\$$	$\mathbf{id} * \mathbf{id}\$$	relacionar $+$
$\mathbf{id} +$	$FT'E'\$$	$\mathbf{id} * \mathbf{id}\$$	emitir $T \to FT'$
$\mathbf{id} +$	$\mathbf{id}\ T'E'\$$	$\mathbf{id} * \mathbf{id}\$$	emitir $F \to \mathbf{id}$
$\mathbf{id} + \mathbf{id}$	$T'E'\$$	$* \mathbf{id}\$$	relacionar $\mathbf{id}$
$\mathbf{id} + \mathbf{id}$	$* FT'E'\$$	$* \mathbf{id}\$$	emitir $T' \to * FT'$
$\mathbf{id} + \mathbf{id} *$	$FT'E'\$$	$\mathbf{id}\$$	relacionar $*$
$\mathbf{id} + \mathbf{id} *$	$\mathbf{id}\ T'E'\$$	$\mathbf{id}\$$	emitir $F \to \mathbf{id}$
$\mathbf{id} + \mathbf{id} * \mathbf{id}$	$T'E'\$$	$\$$	relacionar $\mathbf{id}$
$\mathbf{id} + \mathbf{id} * \mathbf{id}$	$E'\$$	$\$$	emitir $T' \to \epsilon$
$\mathbf{id} + \mathbf{id} * \mathbf{id}$	$\$$	$\$$	emitir $E' \to \epsilon$

Figura 4.21: Movimientos que realiza un analizador sintáctico predictivo con la entrada $\mathbf{id} + \mathbf{id} * \mathbf{id}$

Observe que las formas de frases en esta derivación corresponden a la entrada que ya se ha relacionado (en la columna COINCIDENCIA), seguida del contenido de la pila. La entrada relacionada se muestra sólo para resaltar la correspondencia. Por la misma razón, la parte superior de la pila está a la izquierda; cuando consideremos el análisis sintáctico ascendente, será más natural mostrar la parte superior de la pila a la derecha. El apuntador de la entrada apunta al símbolo por la izquierda de la cadena en la columna ENTRADA. □

4.4.5 Recuperación de errores en el análisis sintáctico predictivo

Esta discusión sobre la recuperación de errores se refiere a la pila de un analizador sintáctico predictivo controlado por una tabla, ya que hace explícitas los terminales y no terminales que el analizador sintáctico espera relacionar con el resto de la entrada; las técnicas también pueden usarse con el análisis sintáctico de descenso recursivo.

Durante el análisis sintáctico predictivo, un error se detecta cuando el terminal en la parte superior de la pila no coincide con el siguiente símbolo de entrada, o cuando el no terminal A se encuentra en la parte superior de la pila, a es el siguiente símbolo de entrada y $M[A, a]$ es **error** (es decir, la entrada en la tabla de análisis sintáctico está vacía).

Modo de pánico

La recuperación de errores en modo de pánico se basa en la idea de omitir símbolos en la entrada hasta que aparezca un token en un conjunto seleccionado de tokens de sincronización. Su

efectividad depende de la elección del conjunto de sincronización. Los conjuntos deben elegirse de forma que el analizador sintáctico se recupere con rapidez de los errores que tengan una buena probabilidad de ocurrir en la práctica. Algunas heurísticas son:

1. Como punto inicial, colocar todos los símbolos que están en SIGUIENTE(A) en el conjunto de sincronización para el no terminal A. Si omitimos tokens hasta que se vea un elemento de SEGUIMENTO(A) y sacamos a A de la pila, es probable que el análisis sintáctico pueda continuar.

2. No basta con usar SIGUIENTE(A) como el conjunto de sincronización para A. Por ejemplo, si los signos de punto y coma terminan las instrucciones, como en C, entonces las palabras reservadas que empiezan las instrucciones no pueden aparecer en el conjunto SIGUIENTE del no terminal que representa a las expresiones. Un punto y coma faltante después de una asignación puede, por lo tanto, ocasionar que se omita la palabra reservada que empieza la siguiente instrucción. A menudo hay una estructura jerárquica en las construcciones en un lenguaje; por ejemplo, las expresiones aparecen dentro de las instrucciones, las cuales aparecen dentro de bloques, y así sucesivamente. Al conjunto de sincronización de una construcción de bajo nivel podemos agregar los símbolos que empiezan las construcciones de un nivel más alto. Por ejemplo, podríamos agregar palabras clave que empiezan las instrucciones a los conjuntos de sincronización para los no terminales que generan las expresiones.

3. Si agregamos los símbolos en PRIMERO(A) al conjunto de sincronización para el no terminal A, entonces puede ser posible continuar con el análisis sintáctico de acuerdo con A, si en la entrada aparece un símbolo que se encuentre en PRIMERO(A).

4. Si un no terminal puede generar la cadena vacía, entonces la producción que deriva a ϵ puede usarse como predeterminada. Al hacer esto se puede posponer cierta detección de errores, pero no se puede provocar la omisión de un error. Este método reduce el número de terminales que hay que considerar durante la recuperación de errores.

5. Si un terminal en la parte superior de la pila no se puede relacionar, una idea simple es sacar el terminal, emitir un mensaje que diga que se insertó el terminal, y continuar con el análisis sintáctico. En efecto, este método requiere que el conjunto de sincronización de un token consista de todos los demás tokens.

Ejemplo 4.36: El uso de los símbolos en PRIMERO y SIGUIENTE como tokens de sincronización funciona razonablemente bien cuando las expresiones se analizan de acuerdo con la gramática usual (4.28). La tabla de análisis sintáctico para esta gramática de la figura 4.17 se repite en la figura 4.22, en donde "sinc" indica los tokens de sincronización obtenidos del conjunto SIGUIENTE de la no terminal en cuestión. Los conjuntos SIGUIENTE para las no terminales se obtienen del ejemplo 4.30.

La tabla en la figura 4.22 debe usarse de la siguiente forma. Si el analizador sintáctico busca la entrada $M[A, a]$ y descubre que está en blanco, entonces se omite el símbolo de entrada a. Si la entrada es "sinc", entonces se saca el no terminal que está en la parte superior de la pila, en un intento por continuar con el análisis sintáctico. Si un token en la parte superior de la pila no coincide con el símbolo de entrada, entonces sacamos el token de la pila, como dijimos antes.

No terminal	Símbolo de entrada					
	id	+	*	(	)	$
E	$E \to TE'$			$E \to TE'$	sinc	sinc
E'		$E' \to +TE'$			$E' \to \epsilon$	$E' \to \epsilon$
T	$T \to FT'$	sinc		$T \to FT'$	sinc	sinc
T'		$T' \to \epsilon$	$T' \to *FT'$		$T' \to \epsilon$	$T' \to \epsilon$
F	$F \to$ **id**	sinc	sinc	$F \to (E)$	sinc	sinc

Figura 4.22: Tokens de sincronización agregados a la tabla de análisis sintáctico de la figura 4.17

Con la entrada errónea)**id** $*$ + **id**, el analizador sintáctico y el mecanismo de recuperación de errores de la figura 4.22 se comporta como en la figura 4.23. □

Pila	Entrada	Comentario
E \$	) **id** $*$ + **id** \$	error, omitir)
E \$	**id** $*$ + **id** \$	**id** está en PRIMERO(E)
TE' \$	**id** $*$ + **id** \$	
$FT'E'$ \$	**id** $*$ + **id** \$	
id $T'E'$\$	**id** $*$ + **id** \$	
$T'E'$ \$	$*$ + **id** \$	
$*FT'E'$ \$	$*$ + **id** \$	
$FT'E'$ \$	+ **id** \$	error, $M[F, +] =$ sinc
$T'E'$ \$	+ **id** \$	Se sacó F
E' \$	+ **id** \$	
$+TE'$ \$	+ **id** \$	
TE' \$	**id** \$	
$FT'E'$ \$	**id** \$	
id $T'E'$ \$	**id** \$	
$T'E'$ \$	\$	
E' \$	\$	
\$	\$	

Figura 4.23: Movimientos de análisis sintáctico y recuperación de errores realizados por un analizador sintáctico predictivo

La discusión anterior sobre la recuperación en modo de pánico no señala el punto importante relacionado con los mensajes de error. El diseñador del compilador debe proporcionar mensajes de error informativos que no sólo describan el error, sino que también llamen la atención hacia el lugar en donde se descubrió el error.

Recuperación a nivel de frase

La recuperación de errores a nivel de frase se implementa llenando las entradas en blanco en la tabla de análisis sintáctico predictivo con apuntadores a rutinas de error. Estas rutinas pueden modificar, insertar o eliminar símbolos en la entrada y emitir mensajes de error apropiados. También pueden sacar de la pila. La alteración de los símbolos de la pila o el proceso de meter nuevos símbolos a la pila es cuestionable por dos razones. En primer lugar, los pasos que realiza el analizador sintáctico podrían entonces no corresponder a la derivación de ninguna palabra en el lenguaje. En segundo lugar, debemos asegurarnos de que no haya posibilidad de un ciclo infinito. Verificar que cualquier acción de recuperación ocasione en un momento dado que se consuma un símbolo de entrada (o que se reduzca la pila si se ha llegado al fin de la entrada) es una buena forma de protegerse contra tales ciclos.

4.4.6 Ejercicios para la sección 4.4

Ejercicio 4.4.1: Para cada una de las siguientes gramáticas, idee analizadores sintácticos predictivos y muestre las tablas de análisis sintáctico. Puede factorizar por la izquierda o eliminar la recursividad por la izquierda de sus gramáticas primero.

a) La gramática del ejercicio 4.2.2(a).

b) La gramática del ejercicio 4.2.2(b).

c) La gramática del ejercicio 4.2.2(c).

d) La gramática del ejercicio 4.2.2(d).

e) La gramática del ejercicio 4.2.2(e).

f) La gramática del ejercicio 4.2.2(g).

!! **Ejercicio 4.4.2:** ¿Es posible, mediante la modificación de la gramática en cualquier forma, construir un analizador sintáctico predictivo para el lenguaje del ejercicio 4.2.1 (expresiones postfijo con el operando a)?

Ejercicio 4.4.3: Calcule PRIMERO y SIGUIENTE para la gramática del ejercicio 4.2.1.

Ejercicio 4.4.4: Calcule PRIMERO y SIGUIENTE para cada una de las gramáticas del ejercicio 4.2.2.

Ejercicio 4.4.5: La gramática $S \rightarrow a\ S\ a \mid a\ a$ genera todas las cadenas de longitud uniforme de as. Podemos idear un analizador sintáctico de descenso recursivo con rastreo hacia atrás para esta gramática. Si elegimos expandir mediante la producción $S \rightarrow a\ a$ primero, entonces sólo debemos reconocer la cadena aa. Por ende, cualquier analizador sintáctico de descenso recursivo razonable probará $S \rightarrow a\ S\ a$ primero.

a) Muestre que este analizador sintáctico de descenso recursivo reconoce las entradas aa, $aaaa$ y $aaaaaaaa$, pero no $aaaaaa$.

!! b) ¿Qué lenguaje reconoce este analizador sintáctico de descenso recursivo?

Los siguientes ejercicios son pasos útiles en la construcción de una gramática en la "Forma Normal de Chomsky" a partir de gramáticas arbitrarias, como se define en el ejercicio 4.4.8.

! Ejercicio 4.4.6: Una gramática es *libre de* ϵ si ningún cuerpo de las producciones es ϵ (a lo cual se le llama *producción* ϵ).

a) Proporcione un algoritmo para convertir cualquier gramática en una gramática libre de ϵ que genere el mismo lenguaje (con la posible excepción de la cadena vacía; ninguna gramática libre de ϵ puede generar a ϵ).

b) Aplique su algoritmo a la gramática $S \rightarrow aSbS \mid bSaS \mid \epsilon$. *Sugerencia:* Primero busque todas las no terminales que sean *anulables*, lo cual significa que generan a ϵ, tal vez mediante una derivación extensa.

! Ejercicio 4.4.7: Una *producción simple* es una producción cuyo cuerpo es una sola no terminal; por ejemplo, una producción de la forma $A \rightarrow A$.

a) Proporcione un algoritmo para convertir cualquier gramática en una gramática libre de ϵ, sin producciones simples, que genere el mismo lenguaje (con la posible excepción de la cadena vacía) *Sugerencia:* Primero elimine las producciones ϵ y después averigüe para qué pares de no terminales A y B se cumple que $A \stackrel{*}{\Rightarrow} B$ mediante una secuencia de producciones simples.

b) Aplique su algoritmo a la gramática (4.1) en la sección 4.1.2.

c) Muestre que, como consecuencia de la parte (a), podemos convertir una gramática en una gramática equivalente que no tenga *ciclos* (derivaciones de uno o más pasos, en los que $A \stackrel{*}{\Rightarrow} A$ para cierta no terminal A).

!! Ejercicio 4.4.8: Se dice que una gramática está en *Forma Normal de Chomsky* (FNC) si toda producción es de la forma $A \rightarrow BC$ o de la forma $A \rightarrow a$, en donde A, B y C son no terminales, y a es un terminal. Muestre cómo convertir cualquier gramática en una gramática FNC para el mismo lenguaje (con la posible excepción de la cadena vacía; ninguna gramática FNC puede generar a ϵ).

! Ejercicio 4.4.9: Todo lenguaje que tiene una gramática libre de contexto puede reconocerse en un tiempo máximo de $O(n^3)$ para las cadenas de longitud n. Una manera simple de hacerlo, conocida como el algoritmo de *Cocke-Younger-Kasami* (o CYK), se basa en la programación dinámica. Es decir, dada una cadena $a_1 a_2 \cdots a_n$, construimos una tabla T de n por n de tal forma que T_{ij} sea el conjunto de no terminales que generen la subcadena $a_i a_{i+1} \cdots a_j$. Si la gramática subyacente está en FNC (vea el ejercicio 4.4.8), entonces una entrada en la tabla puede llenarse en un tiempo $O(n)$, siempre y cuando llenemos las entradas en el orden apropiado: el menor valor de $j - i$ primero. Escriba un algoritmo que llene en forma correcta las entradas de la tabla, y muestre que su algoritmo requiere un tiempo $O(n^3)$. Después de llenar la tabla, ¿cómo podemos determinar si $a_1 a_2 \cdots a_n$ está en el lenguaje?

! **Ejercicio 4.4.10:** Muestre cómo, después de llenar la tabla como en el ejercicio 4.4.9, podemos recuperar en un tiempo $O(n)$ un árbol de análisis sintáctico para $a_1 a_2 \cdots a_n$. *Sugerencia:* Modifique la tabla de manera que registre, para cada no terminal A en cada entrada de la tabla T_{ij}, cierto par de no terminales en otras entradas en la tabla que justifiquen la acción de colocar a A en T_{ij}.

! **Ejercicio 4.4.11:** Modifique su algoritmo del ejercicio 4.4.9 de manera que busque, para cualquier cadena, el menor número de errores de inserción, eliminación y mutación (cada error de un solo carácter) necesarios para convertir la cadena en una cadena del lenguaje de la gramática subyacente.

instr	→	**if** *e* **then** *instr* *colaInstr*
	|	**while** *e* **do** *instr*
	|	**begin** *lista* **end**
	|	*s*
colaInstr	→	**else** *instr*
	|	ϵ
lista	→	*instr colaLista*
colaLista	→	; *lista*
	→	ϵ

Figura 4.24: Una gramática para ciertos tipos de instrucciones

! **Ejercicio 4.4.12:** En la figura 4.24 hay una gramática para ciertas instrucciones. Podemos considerar que *e* y *s* son terminales que representan expresiones condicionales y "otras instrucciones", respectivamente. Si resolvemos el conflicto en relación con la expansión de la instrucción "else" opcional (la no terminal *colaInstr*) al preferir consumir un **else** de la entrada cada vez que veamos uno, podemos construir un analizador sintáctico predictivo para esta gramática. Usando la idea de los símbolos de sincronización descritos en la sección 4.4.5:

a) Construya una tabla de análisis sintáctico predictivo con corrección de errores para la gramática.

b) Muestre el comportamiento de su analizador sintáctico con las siguientes entradas:

(*i*) **if** *e* **then** *s* ; **if** *e* **then** *s* **end**
(*ii*) **while** *e* **do begin** *s* ; **if** *e* **then** *s* ; **end**

4.5 Análisis sintáctico ascendente

Un análisis sintáctico ascendente corresponde a la construcción de un árbol de análisis sintáctico para una cadena de entrada que empieza en las hojas (la parte inferior) y avanza hacia la raíz (la parte superior). Es conveniente describir el análisis sintáctico como el proceso de construcción de árboles de análisis sintáctico, aunque de hecho un front-end de usuario podría realizar una traducción directamente, sin necesidad de construir un árbol explícito. La secuencia

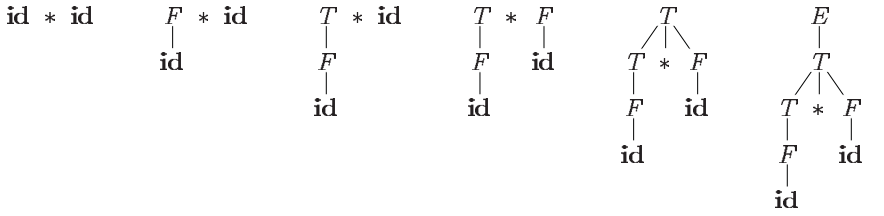

Figura 4.25: Un análisis sintáctico ascendente para **id** $*$ **id**

de imágenes de árboles en la figura 4.25 ilustra un análisis sintáctico ascendente del flujo de tokens **id** $*$ **id**, con respecto a la gramática de expresiones (4.1).

Esta sección presenta un estilo general de análisis sintáctico ascendente, conocido como análisis sintáctico de desplazamiento-reducción. En las secciones 4.6 y 4.7 hablaremos sobre las gramáticas LR, la clase más extensa de gramáticas para las cuales pueden construirse los analizadores sintácticos de desplazamiento-reducción. Aunque es demasiado trabajo construir un analizador sintáctico LR en forma manual, las herramientas conocidas como generadores automáticos de analizadores sintácticos facilitan la construcción de analizadores sintácticos LR eficientes a partir de gramáticas adecuadas. Los conceptos en esta sección son útiles para escribir gramáticas adecuadas que nos permitan hacer un uso efectivo de un generador de analizadores sintácticos LR. En la sección 4.7 aparecen los algoritmos para implementar los generadores de analizadores sintácticos.

4.5.1 Reducciones

Podemos considerar el análisis sintáctico ascendente como el proceso de "reducir" una cadena w al símbolo inicial de la gramática. En cada paso de *reducción*, se sustituye una subcadena específica que coincide con el cuerpo de una producción por el no terminal que se encuentra en el encabezado de esa producción.

Las decisiones clave durante el análisis sintáctico ascendente son acerca de cuándo reducir y qué producción aplicar, a medida que procede el análisis sintáctico.

Ejemplo 4.37: Las imágenes en la figura 4.25 ilustran una secuencia de reducciones; la gramática es la gramática de expresiones (4.1). Hablaremos sobre las reducciones en términos de la siguiente secuencia de cadenas:

$$\textbf{id} * \textbf{id},\ F * \textbf{id},\ T * \textbf{id},\ T * F,\ T,\ E$$

Las cadenas en esta secuencia se forman a partir de las raíces de todos los subárboles de las imágenes. La secuencia empieza con la cadena de entrada **id** $*$ **id**. La primera reducción produce $F * \textbf{id}$ al reducir el **id** por la izquierda a F, usando la producción $F \rightarrow \textbf{id}$. La segunda reducción produce $T * \textbf{id}$ al reducir F a T.

Ahora tenemos una elección entre reducir la cadena T, que es el cuerpo de $E \rightarrow T$, y la cadena que consiste en el segundo **id**, que es el cuerpo de $F \rightarrow \textbf{id}$. En vez de reducir T a E, el segundo **id** se reduce a T, con lo cual se produce la cadena $T * F$. Después, esta cadena se reduce a T. El análisis sintáctico termina con la reducción de T al símbolo inicial E. □

Por definición, una reducción es el inverso de un paso en una derivación (recuerde que en una derivación, un no terminal en una forma de frase se sustituye por el cuerpo de una de sus producciones). Por lo tanto, el objetivo del análisis sintáctico ascendente es construir una derivación en forma inversa. La siguiente derivación corresponde al análisis sintáctico en la figura 4.25:

$$E \Rightarrow T \Rightarrow T * F \Rightarrow T * \mathbf{id} \Rightarrow F * \mathbf{id} \Rightarrow \mathbf{id} * \mathbf{id}$$

Esta derivación es de hecho una derivación por la derecha.

4.5.2 Poda de mangos

Durante una exploración de izquierda a derecha de la entrada, el análisis sintáctico ascendente construye una derivación por la derecha en forma inversa. De manera informal, un "mango" es una subcadena que coincide con el cuerpo de una producción, y cuya reducción representa un paso a lo largo del inverso de una derivación por la derecha.

Por ejemplo, si agregamos subíndices a los tokens **id** para mejorar la legibilidad, los mangos durante el análisis sintáctico de $\mathbf{id}_1 * \mathbf{id}_2$, de acuerdo con la gramática de expresiones (4.1), son como en la figura 4.26. Aunque T es el cuerpo de la producción $E \rightarrow T$, el símbolo T no es un mango en la forma de frase $T * \mathbf{id}_2$. Si T se sustituyera por E, obtendríamos la cadena $E * \mathbf{id}_2$, lo cual no puede derivarse del símbolo inicial E. Por ende, la subcadena por la izquierda que coincide con el cuerpo de alguna producción no necesita ser un mango.

Forma de frase derecha	Mango	Reducción de la producción
$\mathbf{id}_1 * \mathbf{id}_2$	$\mathbf{id}_1$	$F \rightarrow \mathbf{id}$
$F * \mathbf{id}_2$	F	$T \rightarrow F$
$T * \mathbf{id}_2$	$\mathbf{id}_2$	$F \rightarrow \mathbf{id}$
$T * F$	$T * F$	$E \rightarrow T * F$

Figura 4.26: Mangos durante un análisis sintáctico de $\mathbf{id}_1 * \mathbf{id}_2$

De manera formal, si $S \underset{rm}{\overset{*}{\Rightarrow}} \alpha A w \underset{rm}{\Rightarrow} \alpha\beta w$, como en la figura 4.27, entonces la producción $A \rightarrow \beta$ en la posición que sigue después de α es un *mango* de $\alpha\beta w$. De manera alternativa, un mango de la forma de frase derecha γ es una producción $A \rightarrow \beta$ y una posición de γ en donde puede encontrarse la cadena β, de tal forma que al sustituir β en esa posición por A se produzca la forma de frase derecha anterior en una derivación por la derecha de γ.

Observe que la cadena w a la derecha del mango debe contener sólo símbolos terminales. Por conveniencia, nos referimos al cuerpo β en vez de $A \rightarrow \beta$ como un mango. Observe que decimos "un mango" en vez de "el mango", ya que la gramática podría ser ambigua, con más de una derivación por la derecha de $\alpha\beta w$. Si una gramática no tiene ambigüedad, entonces cada forma de frase derecha de la gramática tiene sólo un mango.

Puede obtenerse una derivación por la derecha en forma inversa mediante la "poda de mangos". Es decir, empezamos con una cadena de terminales w a las que se les va a realizar

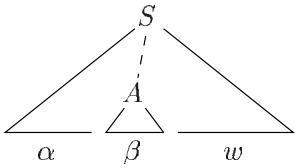

Figura 4.27: Un mango $A \to \beta$ en el árbol de análisis sintáctico para $\alpha\beta w$

el análisis sintáctico. Si w es un enunciado de la gramática a la mano, entonces dejamos que $w = \gamma_n$, en donde γ_n es la n-ésima forma de frase derecha de alguna derivación por la derecha, que todavía se desconoce:

$$S = \gamma_0 \underset{rm}{\Rightarrow} \gamma_1 \underset{rm}{\Rightarrow} \gamma_2 \underset{rm}{\Rightarrow} \cdots \underset{rm}{\Rightarrow} \gamma_{n-1} \underset{rm}{\Rightarrow} \gamma_n = w$$

Para reconstruir esta derivación en orden inverso, localizamos el mango β_n en γ_n y sustituimos β_n por el encabezado de la producción $A_n \to \beta_n$ para obtener la forma de frase derecha γ_{n-1} anterior. Tenga en cuenta que todavía no sabemos cómo se van a encontrar los mangos, pero en breve veremos métodos para hacerlo.

Después repetimos este proceso. Es decir, localizamos el mango β_{n-1} en γ_{n-1} y reducimos este mango para obtener la forma de frase derecha γ_{n-2}. Si al continuar este proceso producimos una forma de frase derecha que consista sólo en el símbolo inicial S, entonces nos detenemos y anunciamos que el análisis sintáctico se completó con éxito. El inverso de la secuencia de producciones utilizadas en las reducciones es una derivación por la derecha para la cadena de entrada.

4.5.3 Análisis sintáctico de desplazamiento-reducción

El análisis sintáctico de desplazamiento-reducción es una forma de análisis sintáctico ascendente, en la cual una pila contiene símbolos gramaticales y un búfer de entrada contiene el resto de la cadena que se va a analizar. Como veremos, el mango siempre aparece en la parte superior de la pila, justo antes de identificarla como el mango.

Utilizamos el \$ para marcar la parte inferior de la pila y también el extremo derecho de la entrada. Por convención, al hablar sobre el análisis sintáctico ascendente, mostramos la parte superior de la pila a la derecha, en vez de a la izquierda como hicimos para el análisis sintáctico descendente. Al principio la pila está vacía, y la cadena w está en la entrada, como se muestra a continuación:

PILA	ENTRADA
\$	w \$

Durante una exploración de izquierda a derecha de la cadena de entrada, el analizador sintáctico desplaza cero o más símbolos de entrada y los mete en la pila, hasta que esté listo para reducir una cadena β de símbolos gramaticales en la parte superior de la pila. Después reduce β al encabezado de la producción apropiada. El analizador sintáctico repite este ciclo hasta que haya detectado un error, o hasta que la pila contenga el símbolo inicial y la entrada esté vacía:

PILA	ENTRADA
\$ S	\$

Al entrar a esta configuración, el analizador sintáctico se detiene y anuncia que el análisis sintáctico se completó con éxito. La figura 4.28 avanza por pasos a través de las acciones que podría realizar un analizador sintáctico de desplazamiento-reducción al analizar la cadena de entrada $\mathbf{id}_1 * \mathbf{id}_2$, de acuerdo con la gramática de expresiones (4.1).

Pila	Entrada	Acción
\$	$\mathbf{id}_1 * \mathbf{id}_2 \$$	desplazar
$\$ \mathbf{id}_1$	$* \mathbf{id}_2 \$$	reducir $F \rightarrow \mathbf{id}$
$\$ F$	$* \mathbf{id}_2 \$$	reducir $T \rightarrow F$
$\$ T$	$* \mathbf{id}_2 \$$	desplazar
$\$ T *$	$\mathbf{id}_2 \$$	desplazar
$\$ T * \mathbf{id}_2$	\$	reducir $F \rightarrow \mathbf{id}$
$\$ T * F$	\$	reducir $T \rightarrow T * F$
$\$ T$	\$	reducir $E \rightarrow T$
$\$ E$	\$	aceptar

Figura 4.28: Configuraciones de un analizador sintáctico de desplazamiento-reducción, con una entrada $\mathbf{id}_1 * \mathbf{id}_2$

Aunque las operaciones primarias son desplazar y reducir, en realidad hay cuatro acciones posibles que puede realizar un analizador sintáctico de desplazamiento-reducción: (1) desplazar, (2) reducir, (3) aceptar y (4) error.

1. *Desplazar*. Desplazar el siguiente símbolo de entrada y lo coloca en la parte superior de la pila.
2. *Reducir*. El extremo derecho de la cadena que se va a reducir debe estar en la parte superior de la pila. Localizar el extremo izquierdo de la cadena dentro de la pila y decidir con qué terminal se va a sustituir la cadena.
3. *Aceptar*. Anunciar que el análisis sintáctico se completó con éxito.
4. *Error*. Descubrir un error de sintaxis y llamar a una rutina de recuperación de errores.

El uso de una pila en el análisis sintáctico de desplazamiento-reducción se justifica debido a un hecho importante: el mango siempre aparecerá en algún momento dado en la parte superior de la pila, nunca en el interior. Este hecho puede demostrarse si consideramos las posibles formas de dos pasos sucesivos en cualquier derivación por la izquierda. La figura 4.29 ilustra los dos posibles casos. En el caso (1), A se sustituye por βBy, y después el no terminal B por la derecha en el cuerpo βBy se sustituye por γ. En el caso (2), A se expande primero otra vez, pero ahora el cuerpo es una cadena y que consiste sólo en terminales. El siguiente no terminal B por la derecha se encontrará en alguna parte a la derecha de y.

En otras palabras:

$$(1) \quad S \overset{*}{\underset{rm}{\Rightarrow}} \alpha Az \underset{rm}{\Rightarrow} \alpha\beta Byz \underset{rm}{\Rightarrow} \alpha\beta\gamma yz$$
$$(2) \quad S \overset{*}{\underset{rm}{\Rightarrow}} \alpha Bx Az \underset{rm}{\Rightarrow} \alpha Bxyz \underset{rm}{\Rightarrow} \alpha\gamma xyz$$

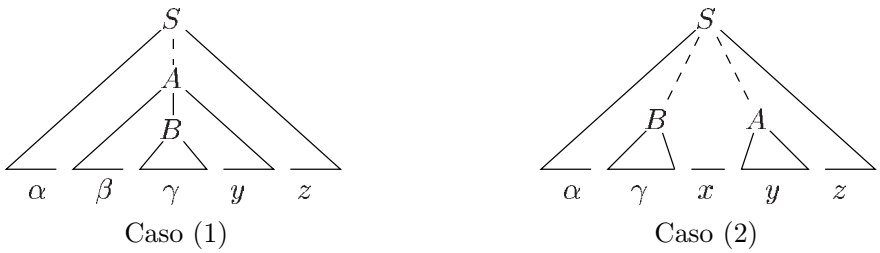

Figura 4.29: Casos para dos pasos sucesivos de una derivación por la derecha

Considere el caso (1) a la inversa, en donde un analizador sintáctico de desplazamiento-reducción acaba de llegar a la siguiente configuración:

PILA	ENTRADA
$\$\alpha\beta\gamma$	$y\,z\$$

El analizador sintáctico reduce el mango γ a B para llegar a la siguiente configuración:

$\$\alpha\beta B$	$y\,z\$$

Ahora el analizador puede desplazar la cadena y y colocarla en la pila, mediante una secuencia de cero o más movimientos de desplazamiento para llegar a la configuración

$\$\alpha\beta By$	$z\$$

con el mango βBy en la parte superior de la pila, y se reduce a A.

Ahora considere el caso (2). En la configuración

$\$\alpha\gamma$	$x\,y\,z\$$

el mango γ está en la parte superior de la pila. Después de reducir el mango γ a B, el analizador sintáctico puede reducir la cadena xy para meter el siguiente mango y en la parte superior de la pila, listo para reducirse a A:

$\$\alpha Bxy$	$z\$$

En ambos casos, después de realizar una reducción, el analizador sintáctico tuvo que desplazar cero o más símbolos para meter el siguiente mango en la pila. Nunca tuvo que buscar el mango dentro de la pila.

4.5.4 Conflictos durante el análisis sintáctico de desplazamiento-reducción

Existen gramáticas libres de contexto para las cuales no se puede utilizar el análisis sintáctico de desplazamiento-reducción. Cada analizador sintáctico de desplazamiento-reducción para una gramática de este tipo puede llegar a una configuración en la cual el analizador sintáctico, conociendo el contenido completo de la pila y el siguiente símbolo de entrada, no puede decidir

si va a desplazar o a reducir (un *conflicto de desplazamiento/reducción*), o no puede decidir qué reducciones realizar (un *conflicto de reducción/reducción*). Ahora veremos algunos ejemplos de construcciones sintácticas que ocasionan tales gramáticas. Técnicamente, estas gramáticas no están en la clase LR(k) de gramáticas definidas en la sección 4.7; nos referimos a ellas como gramáticas no LR. La k en LR(k) se refiere al número de símbolos de preanálisis en la entrada. Las gramáticas que se utilizan en la compilación, por lo general, entran en la clase LR(1), con un símbolo de anticipación a lo más.

Ejemplo 4.38: Una gramática ambigua nunca podrá ser LR. Por ejemplo, considere la gramática del else colgante (4.14) de la sección 4.3:

$$\begin{array}{rcl} \textit{instr} & \rightarrow & \textbf{if } \textit{expr} \textbf{ then } \textit{instr} \\ & | & \textbf{if } \textit{expr} \textbf{ then } \textit{instr} \textbf{ else } \textit{instr} \\ & | & \textbf{otra} \end{array}$$

Si tenemos un analizador sintáctico de desplazamiento-reducción con la siguiente configuración:

PILA	ENTRADA
$\cdots$ **if** *expr* **then** *instr*	**else** $\cdots$ \$

no podemos saber si **if** *expr* **then** *instr* es el mango, sin importar lo que aparezca debajo de él en la pila. Aquí tenemos un conflicto de desplazamiento/reducción. Dependiendo de lo que siga después del **else** en la entrada, podría ser correcto reducir **if** *expr* **then** *instr* a *instr*, o podría ser correcto desplazar el **else** y después buscar otro *instr* para completar la expresión alternativa **if** *expr* **then** *instr* **else** *instr*.

Observe que el análisis sintáctico de desplazamiento-reducción puede adaptarse para analizar ciertas gramáticas ambiguas, como la gramática if-then-else anterior. Si resolvemos el conflicto de desplazamiento/reducción en el **else** a favor del desplazamiento, el analizador sintáctico se comportará como esperamos, asociando cada **else** con el **then** anterior sin coincidencia. En la sección 4.8 hablaremos sobre los analizadores sintácticos para dichas gramáticas ambiguas. □

Otra configuración común para los conflictos ocurre cuando sabemos que tenemos un mango, pero el contenido de la pila y el siguiente símbolo de entrada no son suficientes para determinar qué producción debe usarse en una reducción. El siguiente ejemplo ilustra esta situación.

Ejemplo 4.39: Suponga que tenemos un analizador léxico que devuelve el nombre de token **id** para todos los nombres, sin importar su tipo. Suponga también que nuestro lenguaje invoca a los procedimientos proporcionando sus nombres, con los parámetros rodeados entre paréntesis, y que los arreglos se referencian mediante la misma sintaxis. Como la traducción de los índices en las referencias a arreglos y los parámetros en las llamadas a procedimientos son distintos, queremos usar distintas producciones para generar listas de parámetros e índices actuales. Por lo tanto, nuestra gramática podría tener producciones como las de la figura 4.30 (entre otras).

Una instrucción que empieza con `p(i,j)` aparecería como el flujo de tokens **id**(**id**, **id**) para el analizador sintáctico. Después de desplazar los primeros tres tokens en la pila, un analizador sintáctico de desplazamiento-reducción tendría la siguiente configuración:

(1)	*instr*	→	**id** `(` *lista_parametros* `)`
(2)	*instr*	→	*expr* `:=` *expr*
(3)	*lista_parametros*	→	*lista_parametros* `,` *parámetro*
(4)	*lista_parametros*	→	*parámetro*
(5)	*parametro*	→	**id**
(6)	*expr*	→	**id** `(` *lista_expr* `)`
(7)	*expr*	→	**id**
(8)	*lista_expr*	→	*lista_expr* `,` *expr*
(9)	*lista_expr*	→	*expr*

Figura 4.30: Producciones que implican llamadas a procedimientos y referencias a arreglos

PILA	ENTRADA
··· **id** `(` **id**	`,` **id** `)` ···

Es evidente que el **id** en la parte superior de la pila debe reducirse, pero ¿mediante qué producción? La elección correcta es la producción (5) si `p` es un procedimiento, pero si `p` es un arreglo, entonces es la producción (7). La pila no indica qué información debemos usar en la en la tabla de símbolos que se obtiene a partir de la declaración de `p`.

Una solución es cambiar el token **id** en la producción (1) a **procid** y usar un analizador léxico más sofisticado, que devuelva el nombre de token **procid** cuando reconozca un lexema que sea el nombre de un procedimiento. Para ello se requeriría que el analizador léxico consultara la tabla de símbolos, antes de devolver un token.

Si realizamos esta modificación, entonces al procesar `p(i,j)` el analizador se encontraría en la siguiente configuración:

PILA	ENTRADA
··· **procid** `(` **id**	`,` **id** `)` ···

o en la configuración anterior. En el caso anterior, elegimos la reducción mediante la producción (5); en el último caso mediante la producción (7). Observe cómo el tercer símbolo de la parte superior de la pila determina la reducción que se va a realizar, aun cuando no está involucrado en la reducción. El análisis sintáctico de desplazamiento-reducción puede utilizar información de más adentro en la pila, para guiar el análisis sintáctico. □

4.5.5 Ejercicios para la sección 4.5

Ejercicio 4.5.1: Para la gramática $S \rightarrow 0\ S\ 1 \mid 0\ 1$ del ejercicio 4.2.2(a), indique el mango en cada una de las siguientes formas de frases derechas:

a) 000111.

b) 00S11.

Ejercicio 4.5.2: Repita el ejercicio 4.5.1 para la gramática $S \rightarrow S\ S + \mid S\ S * \mid a$ del ejercicio 4.2.1 y las siguientes formas de frases derechas:

a) $SSS + a * +$.

b) $SS + a * a +$.

c) $aaa * a + +$.

Ejercicio 4.5.3: Proporcione los análisis sintácticos ascendentes para las siguientes cadenas de entrada y gramáticas:

a) La entrada 000111, de acuerdo a la gramática del ejercicio 4.5.1.

b) La entrada $aaa * a++$, de acuerdo a la gramática del ejercicio 4.5.2.

4.6 Introducción al análisis sintáctico LR: SLR (LR simple)

El tipo más frecuente de analizador sintáctico ascendentes en la actualidad se basa en un concepto conocido como análisis sintáctico LR(k); la "L" indica la exploración de izquierda a derecha de la entrada, la "R" indica la construcción de una derivación por la derecha a la inversa, y la k para el número de símbolos de entrada de preanálisis que se utilizan al hacer decisiones del análisis sintáctico. Los casos $k = 0$ o $k = 1$ son de interés práctico, por lo que aquí sólo consideraremos los analizadores sintácticos LR con $k \leq 1$. Cuando se omite (k), se asume que k es 1.

Esta sección presenta los conceptos básicos del análisis sintáctico LR y el método más sencillo para construir analizadores sintácticos de desplazamiento-reducción, llamados "LR Simple" (o SLR). Es útil tener cierta familiaridad con los conceptos básicos, incluso si el analizador sintáctico LR se construye mediante un generador de analizadores sintácticos automático. Empezaremos con "elementos" y "estados del analizador sintáctico"; la salida de diagnóstico de un generador de analizadores sintácticos LR, por lo general, incluye estados del analizador sintáctico, los cuales pueden usarse para aislar las fuentes de conflictos en el análisis sintáctico.

La sección 4.7 introduce dos métodos más complejos (LR canónico y LALR) que se utilizan en la mayoría de los analizadores sintácticos LR.

4.6.1 ¿Por qué analizadores sintácticos LR?

Los analizadores sintácticos LR son controlados por tablas, en forma muy parecida a los analizadores sintácticos LL no recursivos de la sección 4.4.4. Se dice que una gramática para la cual podemos construir una tabla de análisis sintáctico, usando uno de los métodos en esta sección y en la siguiente, es una *gramática LR*. De manera intuitiva, para que una gramática sea LR, basta con que un analizador sintáctico de desplazamiento-reducción de izquierda a derecha pueda reconocer mangos de las formas de frases derechas, cuando éstas aparecen en la parte superior de la pila.

El análisis sintáctico LR es atractivo por una variedad de razones:

- Pueden construirse analizadores sintácticos LR para reconocer prácticamente todas las construcciones de lenguajes de programación para las cuales puedan escribirse gramáticas libres de contexto. Existen gramáticas libres de contexto que no son LR, pero por lo general se pueden evitar para las construcciones comunes de los lenguajes de programación.

- El método de análisis sintáctico LR es el método de análisis sintáctico de desplazamiento reducción sin rastreo hacia atrás más general que se conoce a la fecha, y aún así puede implementarse con la misma eficiencia que otros métodos más primitivos de desplazamiento-reducción (vea las notas bibliográficas).
- Un analizador sintáctico LR puede detectar un error sintáctico tan pronto como sea posible en una exploración de izquierda a derecha de la entrada.
- La clase de gramáticas que pueden analizarse mediante los métodos LR es un superconjunto propio de la clase de gramáticas que pueden analizarse con métodos predictivos o LL. Para que una gramática sea LR(k), debemos ser capaces de reconocer la ocurrencia del lado derecho de una producción en una forma de frase derecha, con k símbolos de entrada de preanálisis. Este requerimiento es mucho menos estricto que para las gramáticas LL(k), en donde debemos ser capaces de reconocer el uso de una producción, viendo sólo los primeros símbolos k de lo que deriva su lado derecho. Por ende, no debe sorprender que las gramáticas LR puedan describir más lenguajes que las gramáticas LL.

La principal desventaja del método LR es que es demasiado trabajo construir un analizador sintáctico LR en forma manual para una gramática común de un lenguaje de programación. Se necesita una herramienta especializada: un generador de analizadores sintácticos LR. Por fortuna, hay varios generadores disponibles, y en la sección 4.9 hablaremos sobre uno de los que se utilizan con más frecuencia: `Yacc`. Dicho generador recibe una gramática libre de contexto y produce de manera automática un analizador para esa gramática. Si la gramática contiene ambigüedades u otras construcciones que sean difíciles de analizar en una exploración de izquierda a derecha de la entrada, entonces el generador de analizadores sintácticos localiza estas construcciones y proporciona mensajes de diagnóstico detallados.

4.6.2 Los elementos y el autómata LR(0)

¿Cómo sabe un analizador sintáctico de desplazamiento-reducción cuándo desplazar y cuándo reducir? Por ejemplo, con el contenido $\$T$ de la pila y el siguiente símbolo de entrada $*$ en la figura 4.28, ¿cómo sabe el analizador sintáctico que la T en la parte superior de la pila no es un mango, por lo cual la acción apropiada es desplazar y no reducir T a E?

Un analizador sintáctico LR realiza las decisiones de desplazamiento-reducción mediante el mantenimiento de estados, para llevar el registro de la ubicación que tenemos en un análisis sintáctico. Los estados representan conjuntos de "elementos". Un *elemento LR(0)* (*elemento*, para abreviar) de una gramática G es una producción de G con un punto en cierta posición del cuerpo. Por ende, la producción $A \rightarrow XYZ$ produce los siguientes cuatro elementos:

$$A \rightarrow \cdot XYZ$$
$$A \rightarrow X \cdot YZ$$
$$A \rightarrow XY \cdot Z$$
$$A \rightarrow XYZ \cdot$$

La producción $A \rightarrow \epsilon$ genera sólo un elemento, $A \rightarrow \cdot$.

De manera intuitiva, un elemento indica qué parte de una producción hemos visto en un punto dado del proceso de análisis sintáctico. Por ejemplo, el elemento $A \rightarrow \cdot XYZ$ indica

Representación de conjuntos de elementos

Un generador de análisis sintáctico que produce un analizador descendente tal vez requiera representar elementos y conjuntos de elementos en una forma conveniente. Un elemento puede representarse mediante un par de enteros, el primero de los cuales es el número de una de las producciones de la gramática subyacente, y el segundo de los cuales es la posición del punto. Los conjuntos de elementos pueden representarse mediante una lista de estos pares. No obstante, como veremos pronto, los conjuntos necesarios de elementos a menudo incluyen elementos de "cierre", en donde el punto se encuentra al principio del cuerpo. Estos siempre pueden reconstruirse a partir de los otros elementos en el conjunto, por lo que no tenemos que incluirlos en la lista.

que esperamos ver una cadena que pueda derivarse de XYZ a continuación en la entrada. El elemento $A \rightarrow X \cdot YZ$ indica que acabamos de ver en la entrada una cadena que puede derivarse de X, y que esperamos ver a continuación una cadena que pueda derivarse de YZ. El elemento $A \rightarrow XYZ \cdot$ indica que hemos visto el cuerpo XYZ y que puede ser hora de reducir XYZ a A.

Una colección de conjuntos de elementos LR(0), conocida como la colección LR(0) *canónica*, proporciona la base para construir un autómata finito determinista, el cual se utiliza para realizar decisiones en el análisis sintáctico. A dicho autómata se le conoce como *autómata LR(0)*.[3] En especial, cada estado del autómata LR(0) representa un conjunto de elementos en la colección LR(0) canónica. El autómata para la gramática de expresiones (4.1), que se muestra en la figura 4.31, servirá como ejemplo abierto para hablar sobre la colección LR(0) canónica para una gramática.

Para construir la colección LR(0) canónica de una gramática, definimos una gramática aumentada y dos funciones, CERRADURA e ir_A. Si G es una gramática con el símbolo inicial S, entonces G', la *gramática aumentada* para G, es G con un nuevo símbolo inicial S' y la producción $S' \rightarrow S$. El propósito de esta nueva producción inicial es indicar al analizador sintáctico cuándo debe dejar de analizar para anunciar la aceptación de la entrada. Es decir, la aceptación ocurre sólo cuando el analizador sintáctico está a punto de reducir mediante $S' \rightarrow S$.

Cerradura de conjuntos de elementos

Si I es un conjunto de elementos para una gramática G, entonces CERRADURA(I) es el conjunto de elementos que se construyen a partir de I mediante las siguientes dos reglas:

1. Al principio, agregar cada elemento en I a *CERRADURA*(I).
2. Si $A \rightarrow \alpha \cdot B\beta$ está en CERRADURA(I) y $B \rightarrow \gamma$ es una producción, entonces agregar el elemento $B \rightarrow \gamma$ a CERRADURA(I), si no se encuentra ya ahí. Aplicar esta regla hasta que no puedan agregarse más elementos nuevos a CERRADURA(I).

[3] Técnicamente, el autómata deja de ser determinista de acuerdo a la definición de la sección 3.6.4, ya que no tenemos un estado muerto, que corresponde al conjunto vacío de elementos. Como resultado, hay ciertos pares estado-entrada para los cuales no existe un siguiente estado.

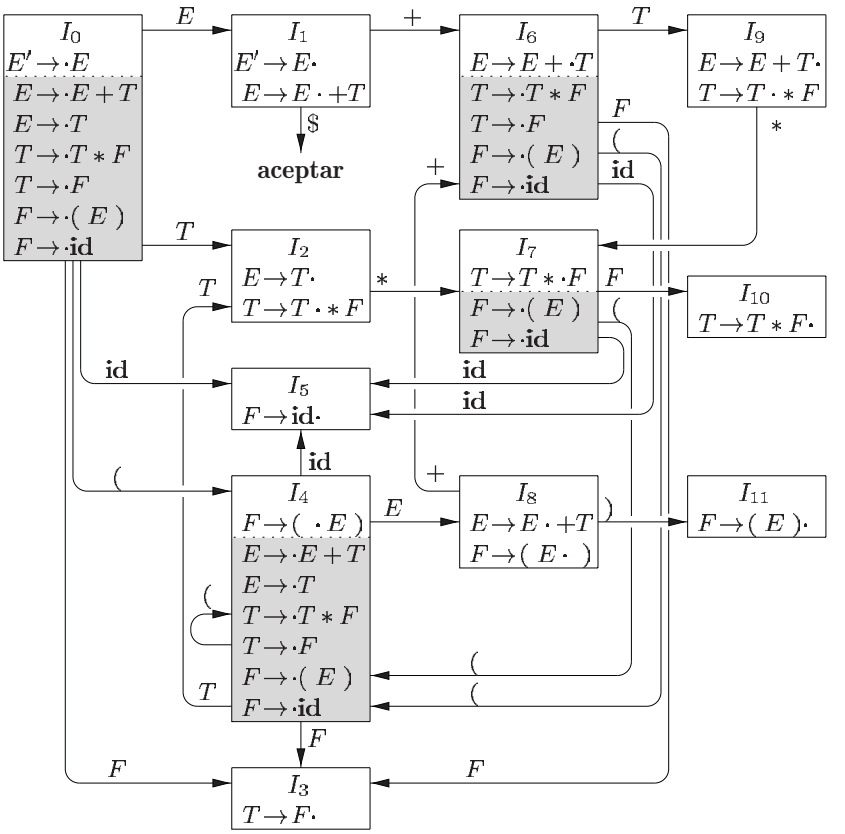

Figura 4.31: Autómata LR(0) para la gramática de expresiones (4.1)

De manera intuitiva, $A \to \alpha \cdot B\beta$ en CERRADURA(I) indica que, en algún punto en el proceso de análisis sintáctico, creemos que podríamos ver a continuación una subcadena que pueda derivarse de $B\beta$ como entrada. La subcadena que pueda derivarse de $B\beta$ tendrá un prefijo que pueda derivarse de B, mediante la aplicación de una de las producciones B. Por lo tanto, agregamos elementos para todas las producciones B; es decir, si $B \to \gamma$ es una producción, también incluimos a $B \to \cdot\gamma$ en CERRADURA(I).

Ejemplo 4.40: Considere la siguiente gramática de expresiones aumentada:

$$\begin{array}{lcl} E' & \to & E \\ E & \to & E + T \mid T \\ T & \to & T * F \mid F \\ E & \to & (E) \mid \mathbf{id} \end{array}$$

Si I es el conjunto de un elemento $\{[E' \to \cdot E]\}$, entonces CERRADURA(I) contiene el conjunto de elementos I_0 en la figura 4.31.

Para ver cómo se calcula la CERRADURA, $E' \rightarrow \cdot E$ se coloca en CERRADURA(I) mediante la regla (1). Como hay una E justo a la derecha de un punto, agregamos las producciones E con puntos en los extremos izquierdos: $E \rightarrow \cdot E + T$ y $E \rightarrow \cdot T$. Ahora hay una T justo a la derecha de un punto en este último elemento, por lo que agregamos $T \rightarrow \cdot T * F$ y $T \rightarrow \cdot F$. A continuación, la F a la derecha de un punto nos obliga a agregar $F \rightarrow \cdot(E)$ y $F \rightarrow \cdot$**id**, pero no necesita agregarse ningún otro elemento. □

La cerradura puede calcularse como en la figura 4.32. Una manera conveniente de implementar la función *cerradura* es mantener un arreglo booleano llamado *agregado*, indexado mediante los no terminales de G, de tal forma que *agregado*[B] se establezca a **true** si, y sólo si agregamos el elemento $B \rightarrow \cdot\gamma$ para cada producción B de la forma $B \rightarrow \gamma$.

```
ConjuntoDeElementos CERRADURA(I) {
    J = I;
    repeat
        for ( cada elemento A → α·Bβ en J )
            for ( cada producción B → γ de G )
                if ( B → ·γ no está en J )
                    agregar B → ·γ a J;
    until no se agreguen más elementos a J en una ronda;
    return J;
}
```

Figura 4.32: Cálculo de CERRADURA

Observe que si se agrega una producción B al cierre de I con el punto en el extremo izquierdo, entonces se agregarán todas las producciones B de manera similar al cierre. Por ende, no es necesario en algunas circunstancias listar los elementos $B \rightarrow \gamma$ que se agregan a I mediante CERRADURA. Basta con una lista de los no terminales B cuyas producciones se agregaron. Dividimos todos los conjuntos de elementos de interés en dos clases:

1. *Elementos del corazón*: el elemento inicial, $S' \rightarrow \cdot S$, y todos los elementos cuyos puntos no estén en el extremo izquierdo.
2. *Elementos que no son del corazón*: todos los elementos con sus puntos en el extremo izquierdo, excepto $S' \rightarrow \cdot S$.

Además, cada conjunto de elementos de interés se forma tomando la cerradura de un conjunto de elementos del corazón; desde luego que los elementos que se agregan en la cerradura nunca podrán ser elementos del corazón. Por lo tanto, podemos representar los conjuntos de elementos en los que realmente estamos interesados con muy poco almacenamiento si descartamos todos los elementos que no sean del corazón, sabiendo que podrían regenerarse mediante el proceso de cerradura. En la figura 4.31, los elementos que no son del corazón se encuentran en la parte sombreada del cuadro para un estado.

La función ir_A

La segunda función útil es ir_A(I, X), en donde I es un conjunto de elementos y X es un símbolo gramatical. ir_A(I, X) se define como la cerradura del conjunto de todos los elementos $[A \rightarrow \alpha X \cdot \beta]$, de tal forma que $[A \rightarrow \alpha \cdot X\beta]$ se encuentre en I. De manera intuitiva, la función ir_A se utiliza para definir las transiciones en el autómata LR(0) para una gramática. Los estados del autómata corresponden a los conjuntos de elementos, y ir_A(I, X) especifica la transición que proviene del estado para I, con la entrada X.

Ejemplo 4.41: Si I es el conjunto de dos elementos $\{[E' \rightarrow E\cdot], [E \rightarrow E\cdot + T]\}$, entonces ir_A($I$, $+$) contiene los siguientes elementos:

$$\begin{array}{lcl} E & \rightarrow & E + \cdot T \\ T & \rightarrow & \cdot T * F \\ T & \rightarrow & \cdot F \\ F & \rightarrow & \cdot(E) \\ F & \rightarrow & \cdot\mathbf{id} \end{array}$$

Para calcular ir_A(I, $+$), examinamos I en busca de elementos con $+$ justo a la derecha del punto. $E' \rightarrow E\cdot$ no es uno de estos elementos, pero $E \rightarrow E\cdot + T$ sí lo es. Desplazamos el punto sobre el $+$ para obtener $E \rightarrow E + \cdot T$ y después tomamos la cerradura de este conjunto singleton. □

Ahora estamos listos para que el algoritmo construya a C, la colección canónica de conjuntos de elementos LR(0) para una gramática aumentada G'; el algoritmo se muestra en la figura 4.33.

```
void elementos(G') {
        C = CERRADURA({[S' → ·S]});
        repeat
                for ( cada conjunto de elementos I en C )
                        for ( cada símbolo gramatical X )
                                if ( ir_A(I, X) no está vacío y no está en C )
                                        agregar ir_A(I, X) a C;
        until no se agreguen nuevos conjuntos de elementos a C en una iteración;
}
```

Figura 4.33: Cálculo de la colección canónica de conjuntos de elementos LR(0)

Ejemplo 4.42: La colección canónica de conjuntos de elementos LR(0) para la gramática (4.1) y la función ir_A se muestran en la figura 4.31. ir_A se codifica mediante las transiciones en la figura. □

Uso del autómata LR(0)

La idea central del análisis sintáctico "LR simple", o SLR, es la construcción del autómata LR(0) a partir de la gramática. Los estados de este autómata son los conjuntos de elementos de la colección LR(0) canónica, y las traducciones las proporciona la función ir_A. El autómata LR(0) para la gramática de expresiones (4.1) apareció antes en la figura 4.31.

El estado inicial del autómata LR(0) es CERRADURA($\{[S' \to \cdot S]\}$), en donde S' es el símbolo inicial de la gramática aumentada. Todos los estados son de aceptaciones. Decimos que el "estado j" se refiere al estado que corresponde al conjunto de elementos I_j.

¿Cómo puede ayudar el autómata LR(0) con las decisiones de desplazar-reducir? Estas decisiones pueden realizarse de la siguiente manera. Suponga que la cadena γ de símbolos gramaticales lleva el autómata LR(0) del estado inicial 0 a cierto estado j. Después, se realiza un desplazamiento sobre el siguiente símbolo de entrada a si el estado j tiene una transición en a. En cualquier otro caso, elegimos reducir; los elementos en el estado j nos indicarán qué producción usar.

El algoritmo de análisis sintáctico LR que presentaremos en la sección 4.6.3 utiliza su pila para llevar el registro de los estados, así como de los símbolos gramaticales; de hecho, el símbolo gramatical puede recuperarse del estado, por lo que la pila contiene los estados. El siguiente ejemplo proporciona una vista previa acerca de cómo pueden utilizarse un autómata LR(0) y una pila de estados para realizar decisiones de desplazamiento-reducción en el análisis sintáctico.

Ejemplo 4.43: La figura 4.34 ilustra las acciones de un analizador sintáctico de desplazamiento-reducción con la entrada **id** $*$ **id**, usando el autómata LR(0) de la figura 4.31. Utilizamos una pila para guardar los estados; por claridad, los símbolos gramaticales que corresponden a los estados en la pila aparecen en la columna SÍMBOLOS. En la línea (1), la pila contiene el estado inicial 0 del autómata; el símbolo correspondiente es el marcador \$ de la parte inferior de la pila.

LÍNEA	PILA	SÍMBOLOS	ENTRADA	ACCIÓN
(1)	0	\$	**id** $*$ **id** \$	desplazar 5
(2)	0 5	\$ **id**	$*$ **id** \$	reducir $F \to$ **id**
(3)	0 3	\$ F	$*$ **id** \$	reducir $T \to F$
(4)	0 2	\$ T	$*$ **id** \$	desplazar 7
(5)	0 2 7	\$ $T*$	**id** \$	desplazar 5
(6)	0 2 7 5	\$ $T*$ **id**	\$	reducir $F \to$ **id**
(7)	0 2 7 10	\$ $T*F$	\$	reducir $T \to T*F$
(8)	0 2	\$ T	\$	reducir $E \to T$
(9)	0 1	\$ E	\$	aceptar

Figura 4.34: El análisis sintáctico de **id** $*$ **id**

El siguiente símbolo de entrada es **id** y el estado 0 tiene una transición en **id** al estado 5. Por lo tanto, realizamos un desplazamiento. En la línea (2), el estado 5 (símbolo **id**) se ha metido en la pila. No hay transición desde el estado 5 con la entrada $*$, por lo que realizamos una reducción. Del elemento $[F \to$ **id**$\cdot]$ en el estado 5, la reducción es mediante la producción $F \to$ **id**.

Con los símbolos, una reducción se implementa sacando el cuerpo de la producción de la pila (en la línea (2), el cuerpo es **id**) y metiendo el encabezado de la producción (en este caso, F). Con los estados, sacamos el estado 5 para el símbolo **id**, lo cual lleva el estado 0 a la parte superior y buscamos una transición en F, el encabezado de la producción. En la figura 4.31, el estado 0 tiene una transición en F al estado 3, por lo que metemos el estado 3, con el símbolo F correspondiente; vea la línea (3).

Como otro ejemplo, considere la línea (5), con el estado 7 (símbolo $*$) en la parte superior de la pila. Este estado tiene una transición al estado 5 con la entrada **id**, por lo que metemos el estado 5 (símbolo **id**). El estado 5 no tiene transiciones, así que lo reducimos mediante $F \rightarrow$ **id**. Al sacar el estado 5 para el cuerpo **id**, el estado 7 pasa a la parte superior de la pila. Como el estado 7 tiene una transición en F al estado 10, metemos el estado 10 (símbolo F). □

4.6.3 El algoritmo de análisis sintáctico LR

En la figura 4.35 se muestra un diagrama de un analizador sintáctico LR. Este diagrama consiste en una entrada, una salida, una pila, un programa controlador y una tabla de análisis sintáctico que tiene dos partes (ACCION y el ir_A). El programa controlador es igual para todos los analizadores sintácticos LR; sólo la tabla de análisis sintáctico cambia de un analizador sintáctico a otro. El programa de análisis sintáctico lee caracteres de un búfer de entrada, uno a la vez. En donde un analizador sintáctico de desplazamiento-reducción desplazaría a un símbolo, un analizador sintáctico LR desplaza a un *estado*. Cada estado sintetiza la información contenida en la pila, debajo de éste.

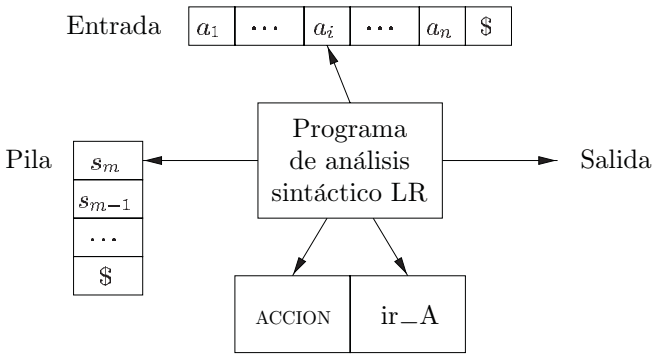

Figura 4.35: Modelo de un analizador sintáctico LR

La pila contiene una secuencia de estados, $s_0 s_1 \cdots s_m$, en donde s_m, se encuentra en la parte superior. En el método SLR, la pila contiene estados del autómata LR(0); los métodos LR canónico y LALR son similares. Por construcción, cada estado tiene un símbolo gramatical correspondiente. Recuerde que los estados corresponden a los conjuntos de elementos, y que hay una transición del estado i al estado j si ir_A$(I_i, X) = I_j$. Todas las transiciones al estado j deben ser para el mismo símbolo gramatical X. Por ende, cada estado, excepto el estado inicial 0, tiene un símbolo gramatical único asociado con él.[4]

[4]Lo opuesto no necesariamente es válido; es decir, más de un estado puede tener el mismo símbolo gramatical. Por

Estructura de la tabla de análisis sintáctico LR

La tabla de análisis sintáctico consiste en dos partes: una función de acción de análisis sintáctico llamada ACCION y una función ir_A.

1. La función ACCION recibe como argumentos un estado i y un terminal a (o \$, el marcador de fin de entrada). El valor de ACCION$[i, a]$ puede tener una de cuatro formas:
   (a) Desplazar j, en donde j es un estado. La acción realizada por el analizador sintáctico desplaza en forma efectiva la entrada a hacia la pila, pero usa el estado j para representar la a.
   (b) Reducir $A \rightarrow \beta$. La acción del analizador reduce en forma efectiva a β en la parte superior de la pila, al encabezado A.
   (c) Aceptar. El analizador sintáctico acepta la entrada y termina el análisis sintáctico.
   (d) Error. El analizador sintáctico descubre un error en su entrada y realiza cierta acción correctiva. En las secciones 4.8.3 y 4.8.4 hablaremos más acerca de cómo funcionan dichas rutinas de recuperación de errores.
2. Extendemos la función ir_A, definida en los conjuntos de elementos, a los estados: si ir_A$[I_i, A] = I_j$, entonces ir_A también asigna un estado i y un no terminal A al estado j.

Configuraciones del analizador sintáctico LR

Para describir el comportamiento de un analizador sintáctico LR, es útil tener una notación que represente el estado completo del analizador sintáctico: su pila y el resto de la entrada. Una *configuración* de un analizador sintáctico LR es un par:

$$(s_0 s_1 \cdots s_m,\ a_i a_{i+1} \cdots a_n\$)$$

en donde el primer componente es el contenido de la pila (parte superior a la derecha) y el segundo componente es el resto de la entrada. Esta configuración representa la forma de frase derecha:

$$X_1 X_2 \cdots X_m a_i a_{i+1} \cdots a_n$$

básicamente en la misma forma en que lo haría un analizador sintáctico de desplazamiento-reducción; la única diferencia es que en vez de símbolos gramaticales, la pila contiene estados a partir de los cuales pueden recuperarse los símbolos gramaticales. Es decir, X_i es el símbolo gramatical representado mediante el estado s_i. Observe que s_0, el estado inicial del analizador sintáctico, no representa a un símbolo gramatical y sirve como marcador de la parte inferior de la pila, así como también juega un papel importante en el análisis sintáctico.

---

ejemplo, vea los estados 1 y 8 en el autómata LR(0) de la figura 4.31, a los cuales se entra mediante las transiciones en E, o los estados 2 y 9, a los cuales se entra mediante las transiciones en T.

Comportamiento del analizador sintáctico LR

El siguiente movimiento del analizador sintáctico a partir de la configuración anterior, se determina mediante la lectura de a_i, el símbolo de entrada actual, y s_m, el estado en la parte superior de la pila, y después se consulta la entrada ACCION$[s_m, a_i]$ en la tabla de acción del análisis sintáctico. Las configuraciones resultantes después de cada uno de los cuatro tipos de movimiento son:

1. Si ACCION$[s_m, a_i]$ = desplazar s, el analizador ejecuta un movimiento de desplazamiento; desplaza el siguiente estado s y lo mete en la pila, introduciendo la siguiente configuración:

   $$(s_0 s_1 \cdots s_m s,\ a_{i+1} \cdots a_n \$)$$

   El símbolo a_i no necesita guardarse en la pila, ya que puede recuperarse a partir de s, si es necesario (en la práctica, nunca lo es). Ahora, el símbolo de entrada actual es a_{i+1}.

2. Si ACCION$[s_m, a_i]$ = reducir $A \rightarrow \beta$, entonces el analizador sintáctico ejecuta un movimiento de reducción, entrando a la siguiente configuración:

   $$(s_0 s_1 \cdots s_{m-r} s,\ a_i a_{i+1} \cdots a_n \$\ )$$

   en donde r es la longitud de β, y $s = \text{ir\_A}[s_{m-r}, A]$. Aquí, el analizador sintáctico primero sacó los símbolos del estado r de la pila, exponiendo al estado s_{m-r}. Después el analizador sintáctico metió a s, la entrada para ir_A$[s_{m-r}, A]$ en la pila. El símbolo de entrada actual no se cambia en un movimiento de reducción. Para los analizadores sintácticos LR que vamos a construir, $X_{m-r+1} \cdots X_m$, la secuencia de los símbolos gramaticales correspondientes a los estados que se sacan de la pila, siempre coincidirá con β, el lado derecho de la producción reductora.

   La salida de un analizador sintáctico LR se genera después de un movimiento de reducción, mediante la ejecución de la acción semántica asociada con la producción reductora. Por el momento, vamos a suponer que la salida consiste sólo en imprimir la producción reductora.

3. Si ACCION$[s_m, a_i]$ = aceptar, se completa el análisis sintáctico.

4. Si ACCION$[s_m, a_i]$ = error, el analizador ha descubierto un error y llama a una rutina de recuperación de errores.

A continuación se sintetiza el algoritmo de análisis sintáctico LR. Todos los analizadores sintácticos LR se comportan de esta manera; la única diferencia entre un analizador sintáctico LR y otro es la información en los campos ACCION e ir_A de la tabla de análisis sintáctico.

Algoritmo 4.44: Algoritmo de análisis sintáctico LR.

ENTRADA: Una cadena de entrada w y una tabla de análisis sintáctico LR con las funciones ACCION e ir_A, para una gramática G.

SALIDA: Si w está en $L(G)$, los pasos de reducción de un análisis sintáctico ascendentes para w; en cualquier otro caso, una indicación de error.

MÉTODO: Al principio, el analizador sintáctico tiene s_0 en su pila, en donde s_0 es el estado inicial y $w\$$ está en el búfer de entrada. Entonces, el analizador ejecuta el programa en la figura 4.36. □

```
hacer que a sea el primer símbolo de w$;
while(1) { /* repetir indefinidamente */
        hacer que s sea el estado en la parte superior de la pila;
        if ( ACCION[s, a] = desplazar t ) {
                meter t en la pila;
                hacer que a sea el siguiente símbolo de entrada;
        } else if ( ACCION[s, a] = reducir A → β ) {
                sacar |β| símbolos de la pila;
                hacer que el estado t ahora esté en la parte superior de la pila;
                meter ir_A[t, A] en la pila;
                enviar de salida la producción A → β;
        } else if ( ACCION[s, a] = aceptar ) break; /* terminó el análisis sintáctico */
        else llamar a la rutina de recuperación de errores;
}
```

Figura 4.36: Programa de análisis sintáctico LR

Ejemplo 4.45: La figura 4.37 muestra las funciones ACCION e ir_A de una tabla de análisis sintáctico LR para la gramática de expresiones (4.1), que repetimos a continuación con las producciones enumeradas:

(1)	$E \rightarrow E + T$	(4)	$T \rightarrow F$
(2)	$E \rightarrow T$	(5)	$T \rightarrow (E)$
(3)	$T \rightarrow T * F$	(6)	$F \rightarrow \mathbf{id}$

Los códigos para las acciones son:

1. si significa desplazar y meter el estado i en la pila,
2. rj significa reducir mediante la producción enumerada como j,
3. acc significa aceptar,
4. espacio en blanco significa error.

Observe que el valor de ir_A$[s, a]$ para el terminal a se encuentra en el campo ACCION conectado con la acción de desplazamiento en la entrada a, para el estado s. El campo ir_A proporciona ir_A$[s, A]$ para los no terminales A. Aunque no hemos explicado aún cómo se seleccionaron las entradas para la figura 4.37, en breve trataremos con esta cuestión.

Estado	Acción						ir_a		
	id	+	*	(	)	$	*E*	*T*	*F*
0	s5			s4			1	2	3
1		s6				acc			
2		r2	s7		r2	r2			
3		r4	r4		r4	r4			
4	s5			s4			8	2	3
5		r6	r6		r6	r6			
6	s5			s4				9	3
7	s5			s4					10
8		s6			s11				
9		r1	s7		r1	r1			
10		r3	r3		r3	r3			
11		r5	r5		r5	r5			

Figura 4.37: Tabla de análisis sintáctico para la gramática de expresiones

En la entrada **id** $*$ **id** $+$ **id**, la secuencia del contenido de la pila y de la entrada se muestra en la figura 4.38. Para fines de claridad, también se muestran las secuencias de los símbolos gramaticales que corresponden a los estados contenidos en la pila. Por ejemplo, en la línea (1) el analizador LR se encuentra en el estado 0, el estado inicial sin símbolo gramatical, y con **id** el primer símbolo de entrada. La acción en la fila 0 y la columna **id** del campo acción de la figura 4.37 es s5, lo cual significa desplazar metiendo el estado 5. Esto es lo que ha ocurrido en la línea (2) se ha metido en la pila el símbolo de estado 5, mientras que **id** se ha eliminado de la entrada.

Después, $*$ se convierte en el símbolo de entrada actual, y la acción del estado 5 sobre la entrada $*$ es reducir mediante $F \rightarrow$ **id**. Se saca un símbolo de estado de la pila. Después se expone el estado 0. Como el ir_A del estado 0 en F es 3, el estado 3 se mete a la pila. Ahora tenemos la configuración de la línea (3). Cada uno de los movimientos restantes se determina en forma similar. □

4.6.4 Construcción de tablas de análisis sintáctico SLR

El método SLR para construir tablas de análisis sintáctico es un buen punto inicial para estudiar el análisis sintáctico LR. Nos referiremos a la tabla de análisis sintáctico construida por este método como una tabla SLR, y a un analizador sintáctico LR que utiliza una tabla de análisis sintáctico SLR como un analizador sintáctico SLR. Los otros dos métodos aumentan el método SLR con información de anticipación.

El método SLR empieza con elementos LR(0) y un autómata LR(0), que presentamos en la sección 4.5. Es decir, dada una gramática G, la aumentamos para producir G', con un nuevo símbolo inicial S'. A partir de G' construimos a C, la colección canónica de conjuntos de elementos para G', junto con la función ir_A.

	PILA	SÍMBOLOS	ENTRADA	ACCIÓN
(1)	0		**id** $*$ **id** $+$ **id** \$	desplazar
(2)	0 5	**id**	$*$ **id** $+$ **id** \$	reducir mediante $F \rightarrow$ **id**
(3)	0 3	F	$*$ **id** $+$ **id** \$	reducir mediante $T \rightarrow F$
(4)	0 2	T	$*$ **id** $+$ **id** \$	desplazar
(5)	0 2 7	$T*$	**id** $+$ **id** \$	desplazar
(6)	0 2 7 5	$T*$ **id**	$+$ **id** \$	reducir mediante $F \rightarrow$ **id**
(7)	0 2 7 10	$T*F$	$+$ **id** \$	reducir mediante $T \rightarrow T*F$
(8)	0 2	T	$+$ **id** \$	reducir mediante $E \rightarrow T$
(9)	0 1	E	$+$ **id** \$	desplazar
(10)	0 1 6	$E+$	**id** \$	desplazar
(11)	0 1 6 5	$E+$ **id**	\$	reducir mediante $F \rightarrow$ **id**
(12)	0 1 6 3	$E+F$	\$	reducir mediante $T \rightarrow F$
(13)	0 1 6 9	$E+T$	\$	reducir mediante $E \rightarrow E+T$
(14)	0 1	E	\$	aceptar

Figura 4.38: Movimientos de un analizador sintáctico LR con **id** $*$ **id** $+$ **id**

Después, las entradas ACCION e ir_A en la tabla de análisis sintáctico se construyen utilizando el siguiente algoritmo. Para ello, requerimos conocer SIGUIENTE(A) para cada no terminal A de una gramática (vea la sección 4.4).

Algoritmo 4.46: Construcción de una tabla de análisis sintáctico SLR.

ENTRADA: Una gramática aumentada G'.

SALIDA: Las funciones ACCION e ir_A para G' de la tabla de análisis sintáctico SLR.

MÉTODO:

1. Construir $C = \{ I_0, I_1, \ldots, I_n \}$, la colección de conjuntos de elementos LR(0) para G'.
2. El estado i se construye a partir de I_i. Las acciones de análisis sintáctico para el estado i se determinan de la siguiente forma:
   (a) Si $[A \rightarrow \alpha \cdot a\beta]$ está en I_i e ir_A(I_i, a) $= I_j$, entonces establecer ACCION[i, a] a "desplazar j". Aquí, a debe ser una terminal.
   (b) Si $[A \rightarrow \alpha \cdot]$ está en I_i, entonces establecer ACCION[i, a] a "reducir $A \rightarrow \alpha$" para toda a en SIGUIENTE(A); aquí, A tal vez no sea S'.
   (c) Si $[S' \rightarrow S \cdot]$ está en I_i, entonces establecer ACCION[i, \$] a "aceptar".

   Si resulta cualquier acción conflictiva debido a las reglas anteriores, decimos que la gramática no es SLR(1). El algoritmo no produce un analizador sintáctico en este caso.

3. Las transiciones de ir_A para el estado i se construyen para todos los no terminales A, usando la regla: Si ir_A(I_i, A) = I_j, entonces ir_A[i, A] = j.
4. Todas las entradas que no estén definidas por las reglas (2) y (3) se dejan como "error".
5. El estado inicial del analizador sintáctico es el que se construyó a partir del conjunto de elementos que contienen $[S' \rightarrow \cdot S]$.

□

A la tabla de análisis sintáctico que consiste en las funciones ACCION e ir_A determinadas por el algoritmo 4.46 se le conoce como la *tabla SLR(1) para G*. A un analizador sintáctico LR que utiliza la tabla SLR(1) para G se le conoce como analizador sintáctico SLR(1) par G, y a una gramática que tiene una tabla de análisis sintáctico SLR(1) se le conoce como *SLR(1)*. Por lo general, omitimos el "(1)" después de "SLR", ya que no trataremos aquí con los analizadores sintácticos que tienen más de un símbolo de preanálisis.

Ejemplo 4.47: Vamos a construir la tabla SLR para la gramática de expresiones aumentada. La colección canónica de conjuntos de elementos LR(0) para la gramática se mostró en la figura 4.31. Primero consideremos el conjunto de elementos I_0:

$$\begin{aligned}
&E' \rightarrow \cdot E\\
&E \rightarrow \cdot E + T\\
&E \rightarrow \cdot T\\
&T \rightarrow \cdot T * F\\
&T \rightarrow \cdot F\\
&F \rightarrow \cdot (E)\\
&F \rightarrow \cdot \mathbf{id}
\end{aligned}$$

El elemento $F \rightarrow \cdot(E)$ produce la entrada ACCION[0, (] = desplazar 4, y el elemento $F \rightarrow \cdot\mathbf{id}$ produce la entrada ACCION[0, **id**] = desplazar 5. Los demás elementos en I_0 no producen ninguna acción. Ahora consideremos I_1:

$$\begin{aligned}
&E' \rightarrow E\cdot\\
&E \rightarrow E\cdot + T
\end{aligned}$$

El primer elemento produce ACCION[1, \$] = aceptar, y el segundo produce ACCION[1, +] = desplazar 6. Ahora consideremos I_2:

$$\begin{aligned}
&E \rightarrow T\cdot\\
&T \rightarrow T\cdot * F
\end{aligned}$$

Como SIGUIENTE(E) = {\$, +,)}, el primer elemento produce lo siguiente:

$$\text{ACCION}[2, \$] = \text{ACCION}[2, +] = \text{ACCION}[2,)] = \text{reducir } E \rightarrow T$$

El segundo elemento produce ACCION[2, $*$] = desplazar 7. Si continuamos de esta forma obtendremos las tablas ACCION y ir_A que se muestran en la figura 4.31. En esa figura, los números de las producciones en las acciones de reducción son los mismos que el orden en el que aparecen en la gramática original (4.1). Es decir, $E \rightarrow E + T$ es el número 1, $E \rightarrow T$ es 2, y así sucesivamente. □

Ejemplo 4.48: Cada una de las gramáticas SLR(1) no tiene ambigüedad, pero hay muchas gramáticas sin ambigüedad que no son SLR(1). Considere la gramática con las siguientes producciones:

$$\begin{array}{lcl} S & \rightarrow & L = R \mid R \\ L & \rightarrow & *R \mid \mathbf{id} \\ R & \rightarrow & L \end{array} \tag{4.49}$$

Consideremos que L y R representan *l-value* y *r-value*, respectivamente, y que $*$ es un operador que indica "el contenido de".[5] La colección canónica de conjuntos de elementos LR(0) para la gramática (4.49) se muestra en la figura 4.39.

I_0: $S' \rightarrow \cdot S$
$S \rightarrow \cdot L = R$
$S \rightarrow \cdot R$
$L \rightarrow \cdot * R$
$L \rightarrow \cdot \mathbf{id}$
$R \rightarrow \cdot L$

I_1: $S' \rightarrow S\cdot$

I_2: $S \rightarrow L\cdot = R$
$R \rightarrow L\cdot$

I_3: $S \rightarrow R\cdot$

I_4: $L \rightarrow *\cdot R$
$R \rightarrow \cdot L$
$L \rightarrow \cdot * R$
$L \rightarrow \cdot \mathbf{id}$

I_5: $L \rightarrow \mathbf{id}\cdot$

I_6: $S \rightarrow L = \cdot R$
$R \rightarrow \cdot L$
$L \rightarrow \cdot * R$
$L \rightarrow \cdot \mathbf{id}$

I_7: $L \rightarrow *R\cdot$

I_8: $R \rightarrow L\cdot$

I_9: $S \rightarrow L = R\cdot$

Figura 4.39: Colección LR(0) canónica para la gramática (4.49)

Considere el conjunto de elementos I_2. El primer elemento en este conjunto hace que ACCION [2, =] sea "desplazar 6". Como SIGUIENTE(R) contiene = (para ver por qué, considere la derivación $S \Rightarrow L = R \Rightarrow *R = R$), el segundo elemento establece ACCION[2, =] a "reducir $R \rightarrow L$". Como hay tanto una entrada de desplazamiento como una de reducción en ACCION[2, =], el estado 2 tiene un conflicto de desplazamiento/reducción con el símbolo de entrada =.

La gramática (4.49) no es ambigua. Este conflicto de desplazamiento/reducción surge del hecho de que el método de construcción del analizador sintáctico SLR no es lo bastante poderoso como para recordar el suficiente contexto a la izquierda para decidir qué acción debe realizar el analizador sintáctico sobre la entrada =, habiendo visto una cadena que puede reducirse a L. Los métodos canónico y LALR, que veremos a continuación, tendrán éxito en una colección más extensa de gramáticas, incluyendo la gramática (4.49). Sin embargo, observe que

[5] Al igual que en la sección 2.8.3, un *l-value* designa una ubicación y un *r-value* es un valor que puede almacenarse en una ubicación.

hay gramáticas sin ambigüedad para las cuales todos los métodos de construcción de analizadores sintácticos LR producirán una tabla de acciones de análisis sintáctico con conflictos. Por fortuna, dichas gramáticas pueden, por lo general, evitarse en las aplicaciones de los lenguajes de programación. □

4.6.5 Prefijos viables

¿Por qué pueden usarse los autómatas LR(0) para realizar decisiones de desplazamiento-reducción? El autómata LR(0) para una gramática caracteriza las cadenas de símbolos gramaticales que pueden aparecer en la pila de un analizador sintáctico de desplazamiento-reducción para la gramática. El contenido de la pila debe ser un prefijo de una forma de frase derecha. Si la pila contiene a α y el resto de la entrada es x, entonces una secuencia de reducciones llevará a αx a S. En términos de derivaciones, $S \underset{rm}{\overset{*}{\Rightarrow}} \alpha x$.

Sin embargo, no todos los prefijos de las formas de frases derechas pueden aparecer en la pila, ya que el analizador sintáctico no debe desplazar más allá del mango. Por ejemplo, suponga que:

$$E \underset{rm}{\overset{*}{\Rightarrow}} F * \mathbf{id} \underset{rm}{\overset{*}{\Rightarrow}} (E) * \mathbf{id}$$

Entonces, en diversos momentos durante el análisis sintáctico, la pila contendrá (, $(E$ y (E), pero no debe contener $(E)*$, ya que (E) es un mango, que el analizador sintáctico debe reducir a F antes de desplazar a $*$.

Los prefijos de las formas de frases derechas que pueden aparecer en la pila de un analizador sintáctico de desplazamiento-reducción se llaman *prefijos viables*. Se definen de la siguiente manera: un prefijo viable es un prefijo de una forma de frase derecha que no continúa más allá del extremo derecho del mango por la derecha de esa forma de frase. Mediante esta definición, siempre es posible agregar símbolos terminales al final de un prefijo viable para obtener una forma de frase derecha.

El análisis sintáctico SLR se basa en el hecho de que los autómatas LR(0) reconocen los prefijos viables. Decimos que el elemento $A \rightarrow \beta_1 \cdot \beta_2$ es *válido* para un prefijo viable $\alpha\beta_1$ si hay una derivación $S' \underset{rm}{\overset{*}{\Rightarrow}} \alpha A w \underset{rm}{\Rightarrow} \alpha\beta_1\beta_2 w$. En general, un elemento será válido para muchos prefijos viables.

El hecho de que $A \rightarrow \beta_1 \cdot \beta_2$ sea válido para $\alpha\beta_1$ nos dice mucho acerca de si debemos desplazar o reducir cuando encontremos a $\alpha\beta_1$ en la pila de análisis sintáctico. En especial, si $\beta_2 \neq \epsilon$, entonces sugiere que no hemos desplazado aún el mango hacia la pila, por lo que el desplazamiento es nuestro siguiente movimiento. Si $\beta_2 = \epsilon$, entonces parece que $A \rightarrow \beta_1$ es el mango, y debemos reducir mediante esta producción. Desde luego que dos elementos válidos pueden indicarnos que debemos hacer distintas cosas para el mismo prefijo viable. Podemos resolver algunos de estos conflictos analizando el siguiente símbolo de entrada, y otros podemos resolverlos mediante los métodos de la sección 4.8, pero no debemos suponer que todos los conflictos de acciones de análisis sintáctico pueden resolverse si se aplica el método LR a una gramática arbitraria.

Podemos calcular con facilidad el conjunto de elementos válidos para cada prefijo viable que puede aparecer en la pila de un analizador sintáctico LR. De hecho, un teorema central de la teoría de análisis sintáctico LR nos dice que el conjunto de elementos válidos para un prefijo viable γ es exactamente el conjunto de elementos a los que se llega desde el estado inicial,

Los elementos como estados de un AFN

Podemos construir un autómata finito no determinista N para reconocer prefijos viables si tratamos a los mismos elementos como estados. Hay una transición desde $A \to \alpha \cdot X\beta$ hacia $A \to \alpha X \cdot \beta$ etiquetada como X, y hay una transición desde $A \to \alpha \cdot B\beta$ hacia $B \to \cdot\gamma$ etiquetada como ϵ. Entonces, CERRADURA(I) para el conjunto de elementos (estados de N) I es exactamente el cierre ϵ de un conjunto de estados de un AFN definidos en la sección 3.7.1. Por ende, ir_A(I, X) proporciona la transición proveniente de I en el símbolo X del AFD construido a partir de N mediante la construcción del subconjunto. Si lo vemos de esta forma, el procedimiento *elementos*(G') en la figura 4.33 es sólo la construcción del mismo subconjunto que se aplica al AFN N, con los elementos como estados.

a lo largo de la ruta etiquetada como γ en el autómata LR(0) para la gramática. En esencia, el conjunto de elementos válidos abarca toda la información útil que puede deducirse de la pila. Aunque aquí no demostraremos este teorema, vamos a ver un ejemplo.

Ejemplo 4.50: Vamos a considerar de nuevo la gramática de expresiones aumentada, cuyos conjuntos de elementos y la función ir_A se exhiben en la figura 4.31. Sin duda, la cadena $E + T*$ es un prefijo viable de la gramática. El autómata de la figura 4.31 se encontrará en el estado 7, después de haber leído $E + T*$. El estado 7 contiene los siguientes elementos:

$$\begin{aligned} T &\to T * \cdot F \\ F &\to \cdot(E) \\ F &\to \cdot\mathbf{id} \end{aligned}$$

que son precisamente los elementos válidos para $E + T*$. Para ver por qué, considere las siguientes tres derivaciones por la derecha:

$$\begin{aligned} E' &\underset{rm}{\Rightarrow} E \\ &\underset{rm}{\Rightarrow} E + T \\ &\underset{rm}{\Rightarrow} E + T * F \end{aligned} \qquad \begin{aligned} E' &\underset{rm}{\Rightarrow} E \\ &\underset{rm}{\Rightarrow} E + T \\ &\underset{rm}{\Rightarrow} E + T * F \\ &\underset{rm}{\Rightarrow} E + T * (E) \end{aligned} \qquad \begin{aligned} E' &\underset{rm}{\Rightarrow} E \\ &\underset{rm}{\Rightarrow} E + T \\ &\underset{rm}{\Rightarrow} E + T * F \\ &\underset{rm}{\Rightarrow} E + T * \mathbf{id} \end{aligned}$$

La primera derivación muestra la validez de $T \to T * \cdot F$, la segunda muestra la validez de $F \to \cdot(E)$, y la tercera muestra la validez de $F \to \cdot\mathbf{id}$. Se puede mostrar que no hay otros elementos válidos para $E + T*$, aunque aquí no demostraremos ese hecho. □

4.6.6 Ejercicios para la sección 4.6

Ejercicio 4.6.1: Describa todos los prefijos viables para las siguientes gramáticas:

a) La gramática $S \to 0\ S\ 1 \mid 0\ 1$ del ejercicio 4.2.2(a).

! b) La gramática $S \rightarrow S\ S + \mid S\ S * \mid a$ del ejercicio 4.2.1.

! c) La gramática $S \rightarrow S\ (\ S\ ) \mid \epsilon$ del ejercicio 4.2.2(c).

Ejercicio 4.6.2: Construya los conjuntos SLR de elementos para la gramática (aumentada) del ejercicio 4.2.1. Calcule la función ir_A para estos conjuntos de elementos. Muestre la tabla de análisis sintáctico para esta gramática. ¿Es una gramática SLR?

Ejercicio 4.6.3: Muestre las acciones de su tabla de análisis sintáctico del ejercicio 4.6.2 sobre la entrada $aa * a+$.

Ejercicio 4.6.4: Para cada una de las gramáticas (aumentadas) del ejercicio 4.2.2(a)-(g):

a) Construya los conjuntos SLR de elementos y su función ir_A.

b) Indique cualquier conflicto de acciones en sus conjuntos de elementos.

c) Construya la tabla de análisis sintáctico SLR, si es que existe.

Ejercicio 4.6.5: Muestre que la siguiente gramática:

$$\begin{array}{lcl} S & \rightarrow & A\ a\ A\ b \mid B\ b\ B\ a \\ A & \rightarrow & \epsilon \\ B & \rightarrow & \epsilon \end{array}$$

es LL(1), pero no SLR(1).

Ejercicio 4.6.6: Muestre que la siguiente gramática:

$$\begin{array}{lcl} S & \rightarrow & S\ A \mid A \\ A & \rightarrow & a \end{array}$$

es SLR(1), pero no LL(1).

!! **Ejercicio 4.6.7:** Considere la familia de gramáticas G_n definidas por:

$$\begin{array}{lcll} S & \rightarrow & A_i\ b_i & \text{para } 1 \leq i \leq n \\ A_i & \rightarrow & a_j\ A_i \mid a_j & \text{para } 1 \leq i, j \leq n \text{ e } i \neq j \end{array}$$

Muestre que:

a) G_n tiene $2n^2 - 2$ producciones.

b) G_n tiene $2n^2 + n^2 + n$ conjuntos de elementos LR(0).

c) G_n es SLR(1).

¿Qué dice este análisis acerca de la extensión que pueden llegar a tener los analizadores sintácticos LR?

! **Ejercicio 4.6.8:** Sugerimos que los elementos individuales pudieran considerarse como estados de un autómata finito no determinista, mientras que los conjuntos de elementos válidos son los estados de un autómata finito determinista (vea el recuadro titulado "Los elementos como estados de un AFN" en la sección 4.6.5). Para la gramática $S \rightarrow S\ S + \mid S\ S * \mid a$ del ejercicio 4.2.1:

a) Dibuje el diagrama de transición (AFN) para los elementos válidos de esta gramática, de acuerdo a la regla que se proporciona en el recuadro antes citado.

b) Aplique la construcción de subconjuntos (Algoritmo 3.20) a su AFN, a partir de (a). ¿Cómo se compara el AFD resultante con el conjunto de elementos LR(0) para la gramática?

!! c) Muestre que en todos los casos, la construcción de subconjuntos que se aplica al AFN que proviene de los elementos válidos para una gramática produce los conjuntos LR(0) de elementos.

! **Ejercicio 4.6.9:** La siguiente es una gramática ambigua:

$$\begin{array}{lcl} S & \rightarrow & A\ S \mid b \\ A & \rightarrow & S\ A \mid a \end{array}$$

Construya para esta gramática su colección de conjuntos de elementos LR(0). Si tratamos de construir una tabla de análisis sintáctico LR para la gramática, hay ciertas acciones en conflicto. ¿Qué son? Suponga que tratamos de usar la tabla de análisis sintáctico eligiendo en forma no determinista una posible acción, cada vez que haya un conflicto. Muestre todas las posibles secuencias de acciones con la entrada *abab*.

4.7 Analizadores sintácticos LR más poderosos

En esta sección vamos a extender las técnicas anteriores de análisis sintáctico LR, para usar un símbolo de preanálisis en la entrada. Hay dos métodos distintos:

1. El método "LR canónico", o simplemente "LR", que utiliza al máximo el (los) símbolo(s) de preanálisis. Este método utiliza un extenso conjunto de elementos, conocidos como elementos LR(1).
2. El método "LR con símbolo de preanálisis" o "LALR(lookahead LR)", que se basa en los conjuntos de elementos LR(0), y tiene mucho menos estados que los analizadores sintácticos comunes, basados en los elementos LR(1). Si introducimos con cuidado lecturas anticipadas en los elementos LR(0), podemos manejar muchas gramáticas más con el método LALR que con el SLR, y construir tablas de análisis sintáctico que no sean más grandes que las tablas SLR. LALR es el método de elección en la mayoría de las situaciones.

Después de presentar ambos métodos, concluiremos con una explicación acerca de cómo compactar las tablas de análisis sintáctico LR para los entornos con memoria limitada.

4.7.1 Elementos LR(1) canónicos

Ahora presentaremos la técnica más general para construir una tabla de análisis sintáctico LR a partir de una gramática. Recuerde que en el método SLR, el estado i llama a la reducción mediante $A \rightarrow \alpha$ si el conjunto de elementos I_i contiene el elemento $[A \rightarrow \alpha\cdot]$ y α se encuentra en SIGUIENTE(A). No obstante, en algunas situaciones cuando el estado i aparece en la parte superior de la pila, el prefijo viable $\beta\alpha$ en la pila es tal que βA no puede ir seguida de a en ninguna forma de frase derecha. Por ende, la reducción mediante $A \rightarrow \alpha$ debe ser inválida con la entrada a.

Ejemplo 4.51: Vamos a reconsiderar el ejemplo 4.48, en donde en el estado 2 teníamos el elemento $R \rightarrow L$, el cual podía corresponder a la $A \rightarrow \alpha$ anterior, y a podía ser el signo =, que se encuentra en SIGUIENTE(R). Por ende, el analizador sintáctico SLR llama a la reducción mediante $R \rightarrow L$ en el estado 2, con = como el siguiente símbolo de entrada (también se llama a la acción de desplazamiento, debido al elemento $S \rightarrow L\cdot{=}R$ en el estado 2). Sin embargo, no hay forma de frase derecha de la gramática en el ejemplo 4.48 que empiece como $R = \cdots$. Por lo tanto, el estado 2, que es el estado correspondiente al prefijo viable L solamente, en realidad no debería llamar a la reducción de esa L a R. □

Es posible transportar más información en el estado, que nos permita descartar algunas de estas reducciones inválidas mediante $A \rightarrow \alpha$. Al dividir estados según sea necesario, podemos hacer que cada estado de un analizador sintáctico LR indique con exactitud qué símbolos de entrada pueden ir después de un mango α para el cual haya una posible reducción a A.

La información adicional se incorpora al estado mediante la redefinición de elementos, para que incluyan un símbolo terminal como un segundo componente. La forma general de un elemento se convierte en $[A \rightarrow \alpha \cdot \beta, a]$, en donde $A \rightarrow \alpha\beta$ es una producción y a es un terminal o el delimitador \$ derecho. A un objeto de este tipo le llamamos *elemento LR(1)*. El 1 se refiere a la longitud del segundo componente, conocido como la *lectura anticipada* del elemento.[6] La lectura anticipada no tiene efecto sobre un elemento de la forma $[A\rightarrow \alpha\cdot\beta, a]$, en donde β no es ϵ, pero un elemento de la forma $[A \rightarrow \alpha\cdot, a]$ llama a una reducción mediante $A \rightarrow \alpha$ sólo si el siguiente símbolo de entrada es a. Por ende, nos vemos obligados a reducir mediante $A \rightarrow \alpha$ sólo con esos símbolos de entrada a para los cuales $[A \rightarrow \alpha\cdot, a]$ es un elemento LR(1) en el estado en la parte superior de la pila. El conjunto de tales as siempre será un subconjunto de SIGUIENTE(A), pero podría ser un subconjunto propio, como en el ejemplo 4.51.

De manera formal, decimos que el elemento LR(1) $[A \rightarrow \alpha\cdot\beta, a]$ es *válido* para un prefijo viable γ si hay una derivación $S \underset{rm}{\overset{*}{\Rightarrow}} \delta A w \underset{rm}{\Rightarrow} \delta\alpha\beta w$, en donde

1. $\gamma = \delta\alpha$, y
2. a es el primer símbolo de w, o w es ϵ y, a es \$.

Ejemplo 4.52: Consideremos la siguiente gramática:

[6] Desde luego que son posibles las lecturas anticipadas que sean cadenas de una longitud mayor a uno, pero no las consideraremos en este libro.

$$S \rightarrow B\ B$$
$$B \rightarrow a\ B \mid b$$

Hay una derivación por la derecha $S \underset{rm}{\overset{*}{\Rightarrow}} aaBab \underset{rm}{\Rightarrow} aaaBab$. Podemos ver que el elemento $[B \rightarrow a \cdot B,\ a]$ es válido para un prefijo viable $\gamma = aaa$, si dejamos que $\delta = aa$, $A = B$, $w = ab$ $\alpha = a$ y $\beta = B$ en la definición anterior. También hay una derivación por la derecha $S \underset{rm}{\overset{*}{\Rightarrow}} BaB \underset{rm}{\Rightarrow} BaaB$. De esta derivación podemos ver que el elemento $[B \rightarrow \alpha \cdot B,\ \$]$ es válido para el prefijo viable Baa. $\square$

4.7.2 Construcción de conjuntos de elementos LR(1)

El método para construir la colección de conjuntos de elementos LR(1) válidos es en esencia el mismo que para construir la colección canónica de conjuntos de elementos LR(0). Sólo necesitamos modificar los dos procedimientos CERRADURA e ir_A.

```
ConjuntoDeElementos CERRADURA(I) {
        repeat
                for ( cada elemento [A → α·Bβ, a] en I )
                        for ( cada producción B → γ en G' )
                                for ( cada terminal b en PRIMERO(βa) )
                                        agregar [B → ·γ, b] al conjunto I;
        until no se agreguen más elementos a I;
        return I;
}

ConjuntoDeElementos ir_A(I, X) {
        inicializar J para que sea el conjunto vacío;
        for ( cada elemento [A → α·Xβ, a] en I )
                agregar el elemento [A → αX·β, a] al conjunto J;
        return CERRADURA(J);
}

void elementos(G') {
        inicializar C a CERRADURA({[S' → ·S, $]});
        repeat
                for ( cada conjunto de elementos I en C )
                        for ( cada símbolo gramatical X )
                                if ( ir_A(I, X) no está vacío y no está en C )
                                        agregar ir_A(I, X) a C;
        until no se agreguen nuevos conjuntos de elementos a C;
}
```

Figura 4.40: Construcción de conjuntos de elementos LR(1) para la gramática G'

Para apreciar la nueva definición de la operación CERRADURA, en especial, por qué b debe estar en PRIMERO(βa), considere un elemento de la forma $[A \to \alpha \cdot B\beta,\ a]$ en el conjunto de elementos válido para cierto prefijo viable γ. Entonces hay una derivación por la derecha $S \underset{rm}{\overset{*}{\Rightarrow}} \delta Aax \underset{rm}{\Rightarrow} \delta\alpha B\beta ax$, en donde $\gamma = \delta\alpha$. Suponga que βax deriva a la cadena de terminales by. Entonces, para cada producción de la forma $B \to \eta$ para cierta η, tenemos la derivación $S \underset{rm}{\overset{*}{\Rightarrow}} \gamma Bby \underset{rm}{\Rightarrow} \gamma\eta by$. Por ende, $[B \to \cdot\eta,\ b]$ es válida para γ. Observe que b puede ser el primer terminal derivado a partir de β, o que es posible que β derive a ϵ en la derivación $\beta ax \underset{rm}{\overset{*}{\Rightarrow}} by$ y, por lo tanto, b puede ser a. Para resumir ambas posibilidades, decimos que b puede ser cualquier terminal en PRIMERO(βax), en donde PRIMERO es la función de la sección 4.4. Observe que x no puede contener la primera terminal de by, por lo que PRIMERO(βax) = PRIMERO(βa). Ahora proporcionaremos la construcción de los conjuntos de elementos LR(1).

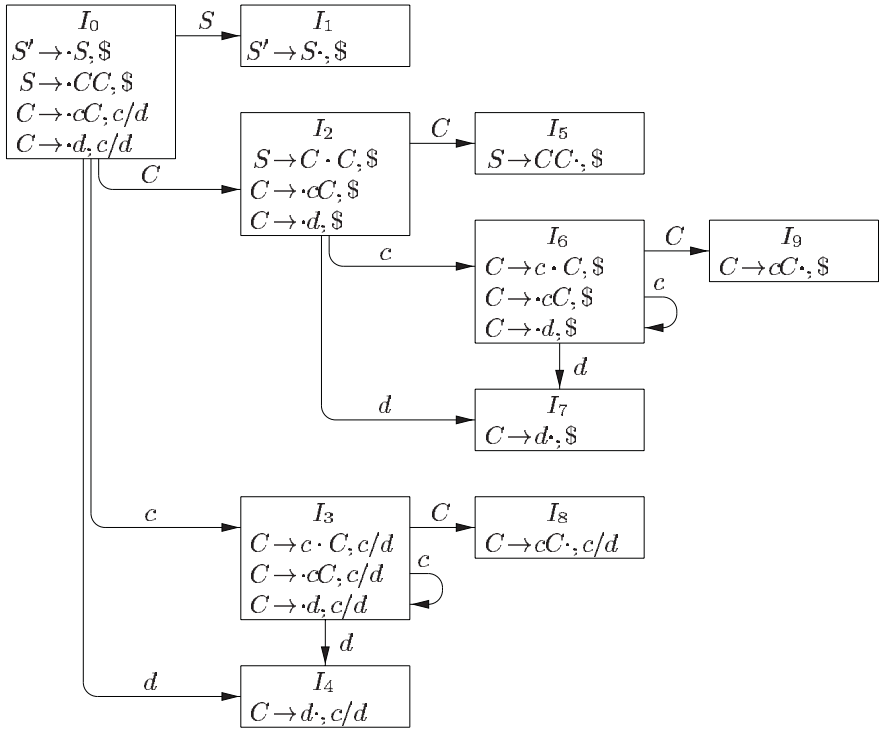

Figura 4.41: El gráfico de ir_A para la gramática (4.55)

Algoritmo 4.53: Construcción de los conjuntos de elementos LR(1).

ENTRADA: Una gramática aumentada G'.

SALIDA: Los conjuntos de elementos LR(1) que son el conjunto de elementos válido para uno o más prefijos viables de G'.

MÉTODO: Los procedimientos CERRADURA e ir_A, y la rutina principal *elementos* para construir los conjuntos de elementos se mostraron en la figura 4.40. □

Ejemplo 4.54: Considere la siguiente gramática aumentada:

$$\begin{array}{rcl} S' & \rightarrow & S \\ S & \rightarrow & C\ C \\ C & \rightarrow & c\ C \mid d \end{array} \tag{4.55}$$

Empezamos por calcular la cerradura de $\{[S' \rightarrow \cdot S, \$]\}$. Relacionamos el elemento $[S' \rightarrow \cdot S, \$]$ con el elemento $[A \rightarrow \alpha \cdot B\beta, a]$ en el procedimiento CERRADURA. Es decir, $A = S'$, $\alpha = \epsilon$, $B = S$, $\beta = \epsilon$ y $a = \$$. La función CERRADURA nos indica que debemos agregar $[B \rightarrow \cdot\gamma, b]$ para cada producción $B \rightarrow y$ y la terminal b en PRIMERO(βa). En términos de la gramática actual, $B \rightarrow \gamma$ debe ser $S \rightarrow CC$, y como β es ϵ y a es \$, b sólo puede ser \$. Por ende, agregamos $[S \rightarrow \cdot CC, \$]$.

Para seguir calculando la cerradura, agregamos todos los elementos $[C \rightarrow \cdot\gamma, b]$ para b en PRIMERO($C\$$). Es decir, si relacionamos $[S \rightarrow \cdot CC, \$]$ con $[A \rightarrow \alpha \cdot B\beta, a]$, tenemos que $A = S$, $\alpha = \epsilon$, $B = C$, $\beta = C$ y $a = \$$. Como C no deriva a la cadena vacía, PRIMERO($C\$$) = PRIMERO(C). Como PRIMERO(C) contiene los terminales c y d, agregamos los elementos $[C \rightarrow \cdot cC, c]$, $[C \rightarrow \cdot cC, d]$, $[C \rightarrow \cdot d, c]$ y $[C \rightarrow \cdot d, d]$. Ninguno de los elementos nuevos tiene un no terminal justo a la derecha del punto, por lo que hemos completado nuestro primer conjunto de elementos LR(1). El conjunto inicial de elementos es:

$$\begin{array}{lll} I_0: & S \rightarrow \cdot S, \$ \\ & S \rightarrow \cdot CC, \$ \\ & C \rightarrow \cdot cC, c/d \\ & C \rightarrow \cdot d, c/d \end{array}$$

Hemos omitido los corchetes por conveniencia de notación, y utilizamos la notación $[C \rightarrow \cdot cC, c/d]$ como abreviación para los dos elementos $[C \rightarrow \cdot cC, c]$ y $[C \rightarrow \cdot cC, d]$.

Ahora calculamos ir_A(I_0, X) para los diversos valores de X. Para $X = S$ debemos la cerradura del elemento $[S' \rightarrow S\cdot, \$]$. No es posible una cerradura adicional, ya que el punto está en el extremo derecho. Por ende, tenemos el siguiente conjunto de elementos:

$$I_1: \quad S' \rightarrow S\cdot, \$$$

Para $X = C$ calculos la cerradura $[S \rightarrow C\cdot C, \$]$. Agregamos las producciones C con el segundo componente \$ y después no podemos agregar más, produciendo lo siguiente:

$$\begin{array}{lll} I_2: & S \rightarrow C\cdot C, \$ \\ & C \rightarrow \cdot cC, \$ \\ & C \rightarrow \cdot d, \$ \end{array}$$

Ahora, dejamos que $X = c$. Debemos calcular la cerradura $\{[C \rightarrow c\cdot C, c/d]\}$. Agregamos las producciones C con el segundo componente c/d, produciendo lo siguiente:

$$\begin{array}{ll} I_3: & C \rightarrow c \cdot C,\ c/d \\ & C \rightarrow \cdot cC,\ c/d \\ & C \rightarrow \cdot d,\ c/d \end{array}$$

Por último, dejamos que $X = d$, y terminamos con el siguiente conjunto de elementos:

$$I_4: \quad C \rightarrow d \cdot,\ c/d$$

Hemos terminado de considerar a ir_A con I_0. No obtenemos nuevos conjuntos de I_1, pero I_2, tiene ir_A en C, c y d. Para ir_A(I_2, C), obtenemos lo siguiente:

$$I_5: \quad S \rightarrow CC \cdot,\ \$$$

sin que se requiera una cerradura. Para calcular ir_A(I_2, c) tomamos la cerradura de $\{[C \rightarrow c \cdot C,\ \$]$, para obtener lo siguiente:

$$\begin{array}{ll} I_6: & C \rightarrow c \cdot C,\ \$ \\ & C \rightarrow \cdot cC,\ \$ \\ & C \rightarrow \cdot d,\ \$ \end{array}$$

Observe que I_6 difiere de I_3 sólo en los segundos componentes. Más adelante veremos que es común para ciertos conjuntos de elementos LR(1) que una gramática tenga los mismos primeros componentes y que difieran en sus segundos componentes. Cuando construyamos la colección de conjuntos de elementos LR(0) para la misma gramática, cada conjunto de LR(0) coincidirá con el conjunto de los primeros componentes de uno o más conjuntos de elementos LR(1). Cuando hablemos sobre el análisis sintáctico LALR, veremos más sobre este fenómeno.

Continuando con la función ir_A para I_2, ir_A(I_2, d) se ve de la siguiente manera:

$$I_7: \quad C \rightarrow d \cdot,\ \$$$

Si pasamos ahora a I_3, los ir_A de I_3 en c y d son I_3 e I_4, respectivamente, y ir_A(I_3, C) es:

$$I_8: \quad C \rightarrow cC \cdot,\ c/d$$

I_4 e I_5 no tienen ir_As, ya que todos los elementos tienen sus puntos en el extremo derecho. Los ir_As de I_6 en c y d son I_6 e I_7, respectivamente, y ir_A(I_6, C) es:

$$I_9: \quad C \rightarrow cC \cdot,\ \$$$

Los conjuntos restantes de elementos no producen mas ir_A, por lo que hemos terminado. La figura 4.41 muestra los diez conjuntos de elementos con sus ir_A. □

4.7.3 Tablas de análisis sintáctico LR(1) canónico

Ahora proporcionaremos las reglas para construir las funciones ACCION e ir_A de LR(1), a partir de los conjuntos de elementos LR(1). Estas funciones se representan mediante una tabla, como antes. La única diferencia está en los valores de las entradas.

Algoritmo 4.56: Construcción de tablas de análisis sintáctico LR canónico.

ENTRADA: Una gramática aumentada G'.

SALIDA: Las funciones ACCION e ir_A de la tabla de análisis sintáctico LR canónico para G'.

MÉTODO:

1. Construir $C' = \{ I_0, I_1, \cdots, I_n \}$, la colección de conjuntos de elementos LR(1) para G'.
2. El estado i del analizador sintáctico se construye a partir de I_i. La acción de análisis sintáctico para el estado i se determina de la siguiente manera.
   (a) Si $[A \to \alpha \cdot a\beta, b]$ está en I_i, e ir_A$(I_i, a) = I_j$, entonces hay que establecer ACCION$[i, a]$ a "desplazar j". Aquí, a debe ser una terminal.
   (b) Si $[A \to \alpha \cdot, a]$ está en I_i, $A \neq S'$, entonces hay que establecer ACCION$[i, a]$ a "reducir $A \to \alpha$".
   (c) Si $[S' \to S \cdot, \$]$ está en I_i, entonces hay que establecer ACCION$[i, \$]$ a "aceptar".

   Si resulta cualquier acción conflictiva debido a las reglas anteriores, decimos que la gramática no es LR(1). El algoritmo no produce un analizador sintáctico en este caso.
3. Las transiciones ir_A para el estado i se construyen para todos los no terminales A usando la regla: Si ir_A$(I_i, A) = I_j$, entonces ir_A$[i, A] = j$.
4. Todas las entradas no definidas por las reglas (2) y (3) se vuelven "error".
5. El estado inicial del analizador sintáctico es el que se construye a partir del conjunto de elementos que contienen $[S' \to \cdot S, \$]$.

□

A la tabla que se forma a partir de la acción de análisis sintáctico y las funciones producidas por el Algoritmo 4.44 se le conoce como la tabla de análisis LR(1) *canónica*. Si la función de acción de análisis sintáctico no tiene entradas definidas en forma múltiple, entonces a la gramática dada se le conoce como *gramática LR(1)*. Como antes, omitimos el "(1)" si queda comprendida su función.

Ejemplo 4.57: La tabla de análisis sintáctico canónica para la gramática (4.55) se muestra en la figura 4.42. Las producciones 1, 2 y 3 son $S \to CC$, $C \to cC$ y $C \to d$, respectivamente. □

Cada gramática SLR(1) es una gramática LR(1), pero para una gramática SLR(1) el analizador sintáctico LR canónico puede tener más estados que el analizador sintáctico SLR para la misma gramática. La gramática de los ejemplos anteriores es SLR, y tiene un analizador sintáctico SLR con siete estados, en comparación con los diez de la figura 4.42.

Estado	Acción			ir_a	
	c	d	\$	S	C
0	s3	s4		1	2
1			acc		
2	s6	s7			5
3	s3	s4			8
4	r3	r3			
5			r1		
6	s6	s7			9
7			r3		
8	r2	r2			
9			r2		

Figura 4.42: Tabla de análisis sintáctico canónica para la gramática (4.55)

4.7.4 Construcción de tablas de análisis sintáctico LALR

Ahora presentaremos nuestro último método de construcción de analizadores sintácticos, la técnica LALR (LR con *lectura anticipada*). Este método se utiliza con frecuencia en la práctica, ya que las tablas que se obtienen son considerablemente menores que las tablas LR canónicas, y a pesar de ello la mayoría de las construcciones sintácticas comunes de los lenguajes de programación pueden expresarse en forma conveniente mediante una gramática LALR. Lo mismo es casi válido para las gramáticas SLR, pero hay algunas construcciones que las técnicas SLR no pueden manejar de manera conveniente (vea el ejemplo 4.48).

Para una comparación del tamaño de los analizadores sintácticos, las tablas SLR y LALR para una gramática siempre tienen el mismo número de estados, y este número consiste, por lo general, en cientos de estados, para un lenguaje como C. La tabla LR canónica tendría, por lo general, varios miles de estados para el lenguaje con el mismo tamaño. Por ende, es mucho más fácil y económico construir tablas SLR y LALR que las tablas LR canónicas.

Con el fin de una introducción, consideremos de nuevo la gramática (4.55), cuyos conjuntos de elementos LR(1) se mostraron en la figura 4.41. Tomemos un par de estados con apariencia similar, como I_4 e I_7. Cada uno de estos estados sólo tiene elementos con el primer componente $C \rightarrow d\cdot$. En I_4, los símbolos de anticipación son c o d; en I_7, \$ es el único símbolo de anticipación.

Para ver las diferencias entre las funciones de I_4 e I_7 en el analizador sintáctico, observe que la gramática genera el lenguaje regular $\mathbf{c^*dc^*d}$. Al leer una entrada $cc\cdots cdcc\cdots cd$, el analizador sintáctico desplaza el primer grupo de cs y la d subsiguiente, y las mete en la pila, entrando al estado 4 después de leer la d. Después, el analizador llama a una reducción mediante $C \rightarrow d$, siempre y cuando el siguiente símbolo de entrada sea c o d. El requerimiento de que sigue c o d tiene sentido, ya que éstos son los símbolos que podrían empezar cadenas en $\mathbf{c^*d}$. Si \$ sigue después de la primera d, tenemos una entrada como ccd, que no está en el lenguaje, y el estado 4 declara en forma correcta un error si \$ es la siguiente entrada.

El analizador sintáctico entra al estado 7 después de leer la segunda d. Después, el analizador debe ver a \$ en la entrada, o de lo contrario empezó con una cadena que no es de la forma

$\mathbf{c}^*\mathbf{dc}^*\mathbf{d}$. Por ende, tiene sentido que el estado 7 deba reducir mediante $C \rightarrow d$ en la entrada \$, y declarar un error en entradas como c o d.

Ahora vamos a sustituir I_4 e I_7 por I_{47}, la unión de I_4 e I_7, que consiste en el conjunto de tres elementos representados por $[C \rightarrow d\cdot,\ c/d/\$]$. Las transacciones ir_A en d que pasan a I_4 o a I_7 desde I_0, I_2, I_3 e I_6 ahora entran a I_{47}. La acción del estado 47 es reducir en cualquier entrada. El analizador sintáctico revisado se comporta en esencia igual que el original, aunque podría reducir d a C en circunstancias en las que el original declararía un error, por ejemplo, en una entrada como *ccd* o *cdcdc*. En un momento dado, el error se atrapará; de hecho, se atrapará antes de que se desplacen más símbolos de entrada.

En forma más general, podemos buscar conjuntos de elementos LR(1) que tengan el mismo *corazón*; es decir, el mismo conjunto de primeros componentes, y podemos combinar estos conjuntos con corazones comunes en un solo conjunto de elementos. Por ejemplo, en la figura 4.41, I_4 e I_7 forman dicho par, con el corazón $\{C \rightarrow d\cdot\}$. De manera similar, I_3 e I_6 forman otro par, con el corazón $\{C \rightarrow c\cdot C,\ C \rightarrow \cdot cC,\ C \rightarrow \cdot d\}$. Hay un par más, I_8 e I_9, con el corazón común $\{C \rightarrow cC\cdot\}$. Observe que, en general, un corazón es un conjunto de elementos LR(0) para la gramática en cuestión, y que una gramática LR(1) puede producir más de dos conjuntos de elementos con el mismo corazón.

Como el corazón de ir_A(I, X) depende sólo del corazón de I, las transacciones ir_A de los conjuntos combinados pueden combinarse entre sí mismos. Por ende, no hay problema al revisar la función ir_A a medida que combinamos conjuntos de elementos. Las funciones activas se modifican para reflejar las acciones sin error de todos los conjuntos de elementos en la combinación.

Suponga que tenemos una gramática LR(1), es decir, una cuyos conjuntos de elementos LR(1) no produzcan conflictos de acciones en el análisis sintáctico. Si sustituimos todos los estados que tengan el mismo corazón con su unión, es posible que la unión resultante tenga un conflicto, pero es poco probable debido a lo siguiente: Suponga que en la unión hay un conflicto en el símbolo anticipado, debido a que hay un elemento $[A \rightarrow \alpha\cdot,\ a]$ que llama a una reducción mediante $A \rightarrow \alpha$, y que hay otro elemento $[B \rightarrow \beta\cdot a\gamma,\ b]$ que llama a un desplazamiento. Entonces, cierto conjunto de elementos a partir del cual se formó la unión tiene el elemento $[A \rightarrow \alpha\cdot,\ a]$, y como los corazones de todos estos estados son iguales, debe tener un elemento $[B \rightarrow \beta\cdot a\gamma,\ c]$ para alguna c. Pero entonces, este estado tiene el mismo conflicto de desplazamiento/reducción en a, y la gramática no era LR(1) como supusimos. Por ende, la combinación de estados con corazones comunes nunca podrá producir un conflicto de desplazamiento/reducción que no haya estado presente en uno de los estados originales, ya que las acciones de desplazamiento dependen sólo del corazón, y no del símbolo anticipado.

Sin embargo, es posible que una combinación produzca un conflicto de reducción/reducción, como se muestra en el siguiente ejemplo.

Ejemplo 4.58: Considere la siguiente gramática:

$$\begin{array}{lcl} S' & \rightarrow & S \\ S & \rightarrow & a\ A\ d \mid b\ B\ d \mid a\ B\ e \mid b\ A\ e \\ A & \rightarrow & c \\ B & \rightarrow & c \end{array}$$

la cual genera las cuatro cadenas *acd*, *ace*, *bcd* y *bce*. El lector puede comprobar que la gramática es LR(1) mediante la construcción de los conjuntos de elementos. Al hacer esto, encontramos

el conjunto de elementos $\{[A \to c\cdot,\ d],\ [B \to c\cdot,\ e]\}$ válido para el prefijo viable ac y $\{[A \to c\cdot,\ e],\ [B \to c\cdot,\ d]\}$ válido para bc. Ninguno de estos conjuntos tiene un conflicto, y sus corazones son iguales. Sin embargo, su unión, que es:

$$\begin{array}{l} A \to c\cdot,\ d/e \\ B \to c\cdot,\ d/e \end{array}$$

genera un conflicto de reducción/reducción, ya que las reducciones mediante $A \to c$ y $B \to c$ se llaman para las entradas d y e. □

Ahora estamos preparados para proporcionar el primero de dos algoritmos de construcción de tablas LALR. La idea general es construir los conjuntos de elementos LR(1), y si no surgen conflictos, combinar los conjuntos con corazones comunes. Después, construiremos la tabla de análisis sintáctico a partir de la colección de conjuntos de elementos combinados. El método que vamos a describir sirve principalmente como una definición de las gramáticas LALR(1). El proceso de construir la colección completa de conjuntos de elementos LR(1) requiere demasiado espacio y tiempo como para que sea útil en la práctica.

Algoritmo 4.59: Una construcción de tablas LALR sencilla, pero que consume espacio.

ENTRADA: Una gramática aumentada G'.

SALIDA: Las funciones de la tabla de análisis sintáctico LALR ACCION e ir_A para G'.

MÉTODO:

1. Construir $C = \{I_0,\ I_1,\ \ldots,\ I_n\}$, la colección de conjuntos de elementos LR(1).
2. Para cada corazón presente entre el conjunto de elementos LR(1), buscar todos los conjuntos que tengan ese corazón y sustituir estos conjuntos por su unión.
3. Dejar que $C' = \{J_0,\ J_1,\ \ldots,\ J_m\}$ sean los conjuntos resultantes de elementos LR(1). Las acciones de análisis sintáctico para el estado i se construyen a partir de J_i, de la misma forma que en el Algoritmo 4.56. Si hay un conflicto de acciones en el análisis sintáctico, el algoritmo no produce un analizador sintáctico y decimos que la gramática no es LALR(1).
4. La tabla ir_A se construye de la siguiente manera. Si J es la unión de uno o más con juntos de elementos LR(1), es decir, $J = I_1 \cap I_2 \cap \cdots \cap I_k$, entonces los corazones de ir_A(I_1, X), ir_A(I_2, X), ..., ir_A(I_k, X) son iguales, ya que $I_1, I_2, \ldots, I_k$ tienen el mismo corazón. Dejar que K sea la unión de todos los conjuntos de elementos que tienen el mismo corazón que ir_A(I_1, X). Entonces, ir_A$(J, X) = K$.

□

A la tabla producida por el algoritmo 4.59 se le conoce como la *tabla de análisis sintáctico LALR* para G. Si no hay conflictos de acciones en el análisis sintáctico, entonces se dice que la gramática dada es una *gramática LALR(1)*. A la colección de conjuntos de elementos que se construye en el paso (3) se le conoce como *colección LALR(1)*.

Ejemplo 4.60: Considere de nuevo la gramática (4.55), cuyo gráfico de ir_A se mostró en la figura 4.41. Como dijimos antes, hay tres pares de conjuntos de elementos que pueden combinarse. I_3 e I_6 se sustituyen por su unión:

$$\begin{array}{lll} I_{36}: & C \rightarrow c \cdot C, & c/d/\$ \\ & C \rightarrow \cdot cC, & c/d/\$ \\ & C \rightarrow \cdot d, & c/d/\$ \end{array}$$

I_4 e I_7 se sustituyen por su unión:

$$I_{47}: \quad C \rightarrow d\cdot, \ c/d/\$$$

después, I_8 e I_9 se sustituyen por su unión:

$$I_{89}: \quad C \rightarrow cC\cdot, \ c/d/\$$$

Las funciones de acción e ir_A LALR para los conjuntos combinados de elementos se muestran en la figura 4.43.

Estado	Acción			ir_A	
	c	d	\$	S	C
0	s36	s47		1	2
1			acc		
2	s36	s47			5
36	s36	s47			89
47	r3	r3	r3		
5			r1		
89	r2	r2	r2		

Figura 4.43: Tabla de análisis sintáctico LALR para la gramática del ejemplo 4.54

Para ver cómo se calculan los ir_A, considere el ir_A(I_{36}, C). En el conjunto original de elementos LR(1), ir_A(I_3, C) = I_8, y ahora I_8 es parte de I_{89}, por lo que hacemos que ir_A(I_{36}, C) sea I_{89}. Podríamos haber llegado a la misma conclusión si consideráramos a I_6, la otra parte de I_{36}. Es decir, ir_A(I_6, C) = I_9, y ahora I_9 forma parte de I_{89}. Para otro ejemplo, considere a ir_A(I_2, c), una entrada que se ejerce después de la acción de desplazamiento de I_2 en la entrada c. En los conjuntos originales de elementos LR(1), ir_A(I_2, c) = I_6. Como I_6 forma ahora parte de I_{36}, ir_A(I_2, c) se convierte en I_{36}. Por lo tanto, la entrada en la figura 4.43 para el estado 2 y la entrada c se convierte en s36, lo cual significa desplazar y meter el estado 36 en la pila. □

Cuando se les presenta una cadena del lenguaje $\mathbf{c^*dc^*d}$, tanto el analizador sintáctico LR de la figura 4.42 como el analizador sintáctico LALR de la figura 4.43 realizan exactamente la misma secuencia de desplazamientos y reducciones, aunque los nombres de los estados en la pila pueden diferir. Por ejemplo, si el analizador sintáctico LR mete a I_3 o I_6 en la pila, el analizador sintáctico LALR meterá a I_{36} en la pila. Esta relación se aplica en general para

una gramática LALR. Los analizadores sintácticos LR y LALR se imitarán uno al otro en las entradas correctas.

Si se le presenta una entrada errónea, el analizador sintáctico LALR tal vez proceda a realizar ciertas reducciones, una vez que el analizador sintáctico LR haya declarado un error. Sin embargo, el analizador sintáctico LALR nunca desplazará otro símbolo después de que el analizador sintáctico LR declare un error. Por ejemplo, en la entrada *ccd* seguida de \$, el analizador sintáctico LR de la figura 4.42 meterá lo siguiente en la pila:

0 3 3 4

y en el estado 4 descubrirá un error, ya que \$ es el siguiente símbolo de entrada y el estado 4 tiene una acción de error en \$. En contraste, el analizador sintáctico LALR de la figura 4.43 realizará los movimientos correspondientes, metiendo lo siguiente en la pila:

0 36 36 47

Pero el estado 47 en la entrada \$ tiene la acción de reducir $C \rightarrow d$. El analizador sintáctico LALR cambiará, por lo tanto, su pila a:

0 36 36 89

Ahora, la acción del estado 89 en la entrada \$ es reducir $C \rightarrow cC$. La pila se convierte en lo siguiente:

0 36 89

en donde se llama a una reducción similar, con lo cual se obtiene la pila:

0 2

Por último, el estado 2 tiene una acción de error en la entrada \$, por lo que ahora se descubre el error.

4.7.5 Construcción eficiente de tablas de análisis sintáctico LALR

Hay varias modificaciones que podemos realizar al Algoritmo 4.59 para evitar construir la colección completa de conjuntos de elementos LR(1), en el proceso de crear una tabla de análisis sintáctico LALR(1).

- En primer lugar, podemos representar cualquier conjunto de elementos I LR(0) o LR(1) mediante su corazón (kernel); es decir, mediante aquellos elementos que sean el elemento inicial, $[S' \rightarrow \cdot S]$ o $[S' \rightarrow \cdot S, \$]$, o que tengan el punto en algún lugar que no sea al principio del cuerpo de la producción.
- Podemos construir los corazones de los elementos LALR(1) a partir de los corazones de los elementos LR(0), mediante un proceso de propagación y generación espontánea de lecturas adelantadas, lo cual describiremos en breve.
- Si tenemos los corazones LALR(1), podemos generar la tabla de análisis sintáctico LALR(1) cerrando cada corazón, usando la función CERRADURA de la figura 4.40, y después calculando las entradas en la tabla mediante el Algoritmo 4.56, como si los conjuntos de elementos LALR(1) fueran conjuntos de elementos LR(1) canónicos.

Ejemplo 4.61: Vamos a usar, como un ejemplo del eficiente método de construcción de una tabla LALR(1), la gramática que no es SLR del ejemplo 4.48, la cual reproducimos a continuación en su forma aumentada:

$$\begin{array}{rcl} S' & \rightarrow & S \\ S & \rightarrow & L = R \mid R \\ L & \rightarrow & *R \mid \mathbf{id} \\ R & \rightarrow & L \end{array}$$

En la figura 4.39 se mostraron los conjuntos completos de elementos LR(0) para esta gramática. Los corazones de estos elementos se muestran en la figura 4.44. □

I_0: $S' \rightarrow \cdot S$

I_1: $S' \rightarrow S\cdot$

I_2: $S \rightarrow L\cdot = R$
$R \rightarrow L\cdot$

I_3: $S \rightarrow R\cdot$

I_4: $L \rightarrow *\cdot R$

I_5: $L \rightarrow \mathbf{id}\cdot$

I_6: $S \rightarrow L = \cdot R$

I_7: $L \rightarrow *R\cdot$

I_8: $R \rightarrow L\cdot$

I_9: $S \rightarrow L = R\cdot$

Figura 4.44: Corazones de los conjuntos de elementos LR(0) para la gramática (4.49)

Ahora debemos adjuntar los símbolos de anticipación apropiados para los elementos LR(0) en los corazones, para crear los corazones de los conjuntos de elementos LALR(1). Hay dos formas en que se puede adjuntar un símbolo de anticipación b a un elemento LR(0) $B \rightarrow \gamma\cdot\delta$, en cierto conjunto de elementos LALR(1) J:

1. Hay un conjunto de elementos I, con un elemento de corazón $A \rightarrow \alpha\cdot\beta$, a, y $J = \text{ir\_A}(I, X)$, y la construcción de

   $$\text{ir\_A}(\text{CERRADURA}(\{[A \rightarrow \alpha\cdot\beta, a]\}), X)$$

   como se proporciona en la figura 4.40, contiene $[B \rightarrow \gamma\cdot\delta, b]$, sin importar a. Se considera que dicho símbolo de anticipación b se genera en forma *espontánea* para $B \rightarrow \gamma\cdot\delta$.

2. Como caso especial, el símbolo de anticipación \$ se genera en forma espontánea para el elemento $S' \rightarrow \cdot S$ en el conjunto inicial de elementos.

3. Todo es como en (1), pero $a = b$ e $\text{ir\_A}(\text{CERRADURA}(\{[A \rightarrow \alpha\cdot\beta, b]\}), X)$, como se proporciona en la figura 4.40, contiene $[B \rightarrow \gamma\cdot\delta, b]$ sólo porque $A \rightarrow \alpha\cdot\beta$ tiene a b como uno de sus símbolos de anticipación asociados. En tal caso, decimos que los símbolos de anticipación se *propagan* desde $A \rightarrow \alpha\cdot\beta$ en el corazón de I, hasta $B \rightarrow \gamma\cdot\delta$ en el corazón de J. Observe que la propagación no depende del símbolo de anticipación específico; o todos los símbolos de anticipación se propagan desde un elemento hasta otro, o ninguno lo hace.

Debemos determinar los símbolos de anticipación generados en forma espontánea para cada conjunto de elementos LR(0), y también determinar cuáles elementos propagan los símbolos de anticipación desde cuáles otros. En realidad, la prueba es bastante simple. Hagamos que # sea un símbolo que no esté en la gramática en cuestión. Hagamos que $A \to \alpha{\cdot}\beta$ sea un elemento de corazón LR(0) en el conjunto I. Calcule, para cada X, $J = \text{ir\_A}(\text{CERRADURA}(\{[A \to \alpha{\cdot}\beta, \#]\}), X)$. Para cada elemento de corazón en J, examinamos su conjunto de símbolos de anticipación. Si # es un símbolo de anticipación, entonces los símbolos de anticipación se propagan hacia ese elemento desde $A \to \alpha{\cdot}\beta$. Cualquier otro símbolo de anticipación se genera de manera espontánea. Estas ideas se hacen precisas en el siguiente algoritmo, el cual también hace uso del hecho de que los únicos elementos de corazón en J deben tener a X justo a la izquierda del punto; es decir, deben ser de la forma $B \to \gamma X{\cdot}\delta$.

Algoritmo 4.62: Determinación de los símbolos de anticipación.

ENTRADA: El corazón K de un conjunto de elementos LR(0) I y un símbolo gramatical X.

SALIDA: Los símbolos de anticipación generados en forma espontánea por los elementos en I, para los elementos de corazón en ir_A(I, X) y los elementos en I, a partir de los cuales los símbolos de anticipación se propagan hacia los elementos de corazón en ir_A(I, X).

MÉTODO: El algoritmo se proporciona en la figura 4.45. □

```
for ( cada elemento A → α·β en K ) {
        J := CERRADURA({[A → α·β, #]} );
        if ( [B → γ·Xδ, a] está en J, y a no es # )
                concluir que el símbolo de anticipación a se genera en forma espontánea
                    para el elemento B → γX·δ en ir_A(I, X);
        if ( [B → γ·Xδ, #] está en J )
                concluir que los símbolos de anticipación se propagan desde A → α·β en I
                    hacia B → γX·δ en ir_A(I, X);
}
```

Figura 4.45: Descubrimiento de símbolos de anticipación propagados y espontáneos

Ahora estamos listos para adjuntar los símbolos de anticipación a los corazones de los conjuntos de elementos LR(0) para formar los conjuntos de elementos LALR(1). En primer lugar, sabemos que \$ es un símbolo de anticipación para $S' \to {\cdot}S$ en el conjunto inicial de elementos LR(0). El Algoritmo 4.62 nos proporciona todos los símbolos de anticipación que se generan en forma espontánea. Después de listar todos esos símbolos de anticipación, debemos permitir que se propaguen hasta que ya no puedan hacerlo más. Hay muchos métodos distintos, todos los cuales en cierto sentido llevan la cuenta de los "nuevos" símbolos de anticipación que se han propagado en un elemento, pero que todavía no se propagan hacia fuera. El siguiente algoritmo describe una técnica para propagar los símbolos de anticipación hacia todos los elementos.

Algoritmo 4.63: Cálculo eficiente de los corazones de la colección de conjuntos de elementos LALR(1).

ENTRADA: Una gramática aumentada G'.

SALIDA: Los corazones de la colección de conjuntos de elementos LALR(1) para G'.

MÉTODO:

1. Construir los corazones de los conjuntos de elementos LR(0) para G. Si el espacio no es de extrema importancia, la manera más simple es construir los conjuntos de elementos LR(0), como en la sección 4.6.2, y después eliminar los elementos que no sean del corazón. Si el espacio está restringido en extremo, tal vez sea conveniente almacenar sólo los elementos del corazón de cada conjunto, y calcular ir_A para un conjunto de elementos I, para lo cual primero debemos calcular la cerradura de I.

2. Aplicar el Algoritmo 4.62 al corazón de cada conjunto de elementos LR(0) y el símbolo gramatical X para determinar qué símbolos de anticipación se generan en forma espontánea para los elementos del corazón en ir_A(I, X), y a partir los cuales se propagan los elementos en los símbolos de anticipación I a los elementos del corazón en ir_A(I, X).

3. Inicializar una tabla que proporcione, para cada elemento del corazón en cada conjunto de elementos, los símbolos de anticipación asociados. Al principio, cada elemento tiene asociados sólo los símbolos de anticipación que determinamos en el paso (2) que se generaron en forma espontánea.

4. Hacer pasadas repetidas sobre los elementos del corazón en todos los conjuntos. Al visitar un elemento i, buscamos los elementos del corazón para los cuales i propaga sus símbolos de anticipación, usando la información que se tabuló en el paso (2). El conjunto actual de símbolos de anticipación para i se agrega a los que ya están asociados con cada uno de los elementos para los cuales i propaga sus símbolos de anticipación. Continuamos realizando pasadas sobre los elementos del corazón hasta que no se propaguen más símbolos nuevos de anticipación.

□

Ejemplo 4.64: Vamos a construir los corazones de los elementos LALR(1) para la gramática del ejemplo 4.61. Los corazones de los elementos LR(0) se mostraron en la figura 4.44. Al aplicar el Algoritmo 4.62 al corazón del conjunto de elementos I_0, primero calculamos CERRADURA($\{[S' \rightarrow \cdot S, \#]\}$), que viene siendo:

$$\begin{array}{ll} S' \rightarrow \cdot S, \# & L \rightarrow \cdot * R, \#/= \\ S \rightarrow \cdot L = R, \# & L \rightarrow \cdot \mathbf{id}, \#/= \\ S \rightarrow \cdot R, \# & R \rightarrow \cdot L, \# \end{array}$$

Entre los elementos en la cerradura, vemos dos en donde se ha generado el símbolo de anticipación = en forma espontánea. El primero de éstos es $L \rightarrow \cdot * R$. Este elemento, con $*$ a la derecha del punto, produce $[L \rightarrow *\cdot R, =]$. Es decir, = es un símbolo de anticipación generado en forma espontánea para $L \rightarrow *\cdot R$, que se encuentra en el conjunto de elementos I_4. De manera similar, $[L \rightarrow \cdot\mathbf{id}, =]$ nos indica que = es un símbolo de anticipación generado en forma espontánea para $L \rightarrow \mathbf{id}\cdot$ en I_5.

Como # es un símbolo de anticipación para los seis elementos en la cerradura, determinamos que el elemento $S' \rightarrow S$ en I_0 propaga los símbolos de anticipación hacia los siguientes seis elementos:

$S' \to S\cdot$ in I_1 $\quad L \to *\cdot R$ in I_4
$S \to L\cdot = R$ in I_2 $\quad L \to \mathbf{id}\cdot$ in I_5
$S \to R\cdot$ in I_3 $\quad R \to L\cdot$ in I_2

Desde		Hacia	
I_0:	$S' \to \cdot S$	I_1:	$S' \to S\cdot$
		I_2:	$S \to L\cdot = R$
		I_2:	$R \to L\cdot$
		I_3:	$S \to R\cdot$
		I_4:	$L \to *\cdot R$
		I_5:	$L \to \mathbf{id}\cdot$
I_2:	$S \to L\cdot = R$	I_6:	$S \to L = \cdot R$
I_4:	$L \to *\cdot R$	I_4:	$L \to *\cdot R$
		I_5:	$L \to \mathbf{id}\cdot$
		I_7:	$L \to *R\cdot$
		I_8:	$R \to L\cdot$
I_6:	$S \to L = \cdot R$	I_4:	$L \to *\cdot R$
		I_5:	$L \to \mathbf{id}\cdot$
		I_8:	$R \to L\cdot$
		I_9:	$S \to L = R\cdot$

Figura 4.46: Propagación de los símbolos de anticipación

En la figura 4.47, mostramos los pasos (3) y (4) del Algoritmo 4.63. La columna etiquetada como INIC muestra los símbolos de anticipación generados en forma espontánea para cada elemento del corazón. Éstas son sólo las dos ocurrencias de = que vimos antes, y el símbolo de anticipación \$ espontáneo para el elemento inicial $S' \to \cdot S$.

En la primera pasada, el símbolo \$ de anticipación se propaga de $S' \to S$ en I_0 hacia los seis elementos listados en la figura 4.46. El símbolo = de anticipación se propaga desde $L \to *\cdot R$ en I_4 hacia los elementos $L \to * R\cdot$ en I_7 y $R \to L\cdot$ en I_8. También se propaga hacia sí mismo y hacia $L \to \mathbf{id}\cdot$ en I_5, pero estos símbolos de anticipación ya están presentes. En la segunda y tercera pasada, el único símbolo nuevo de anticipación que se propaga es \$, descubierto para los sucesores de I_2 e I_4 en la pasada 2 y para el sucesor de I_6 en la pasada 3. En la pasada 4 no se propagan nuevos símbolos de anticipación, por lo que el conjunto final de símbolos de anticipación se muestra en la columna por la derecha de la figura 4.47.

Observe que el conflicto de desplazamiento/reducción en el ejemplo 4.48 que utiliza el método SLR ha desaparecido con la técnica LALR. La razón es que sólo el símbolo \$ de anticipación está asociado con $R \to L\cdot$ en I_2, por lo que no hay conflicto con la acción de análisis sintáctico de desplazamiento en =, generada por el elemento $S \to L\cdot{=}R$ en I_2. $\square$

Conjunto	Elemento	Símbolos de anticipación			
		Inic	Pasada 1	Pasada 2	Pasada 3
I_0:	$S' \to \cdot S$	\$	\$	\$	\$
I_1:	$S' \to S\cdot$		\$	\$	\$
I_2:	$S \to L\cdot = R$		\$	\$	\$
	$R \to L\cdot$		\$	\$	\$
I_3:	$S \to R\cdot$		\$	\$	\$
		=	=/\$	=/\$	=/\$
I_5:	$L \to \mathbf{id}\cdot$	=	=/\$	=/\$	=/\$
I_6:	$S \to L = \cdot R$			\$	\$
I_7:	$L \to *R\cdot$		=	=/\$	=/\$
I_8:	$R \to L\cdot$		=	=/\$	=/\$
I_9:	$S \to L = R\cdot$				\$

Figura 4.47: Cálculo de los símbolos de anticipación

4.7.6 Compactación de las tablas de análisis sintáctico LR

Una gramática de lenguaje de programación común, con una cantidad de 50 a 100 terminales y 100 producciones, puede tener una tabla de análisis sintáctico LALR con varios cientos de estados. La función de acción podría fácilmente tener 20 000 entradas, cada una requiriendo por lo menos 8 bits para codificarla. En los dispositivos pequeños, puede ser importante tener una codificación más eficiente que un arreglo bidimensional. En breve mencionaremos algunas técnicas que se han utilizado para comprimir los campos ACCION e ir_A de una tabla de análisis sintáctico LR.

Una técnica útil para compactar el campo de acción es reconocer que, por lo general, muchas filas de la tabla de acciones son idénticas. Por ejemplo, en la figura 4.42 los estados 0 y 3 tienen entradas de acción idénticas, al igual que los estados 2 y 6. Por lo tanto, podemos ahorrar una cantidad de espacio considerable, con muy poco costo en relación con el tiempo, si creamos un apuntador para cada estado en un arreglo unidimensional. Los apuntadores para los estados con las mismas acciones apuntan a la misma ubicación. Para acceder a la información desde este arreglo, asignamos a cada terminal un número desde cero hasta uno menos que el número de terminales, y utilizamos este entero como un desplazamiento a partir del valor del apuntador para cada estado. En un estado dado, la acción de análisis sintáctico para la i-ésima terminal se encontrará a i ubicaciones más allá del valor del apuntador para ese estado.

Puede lograrse una mejor eficiencia en cuanto al espacio, a expensas de un analizador sintáctico un poco más lento, mediante la creación de una lista para las acciones de cada estado. Esta lista consiste en pares (terminal-símbolo, acción). La acción más frecuente para un estado puede

colocarse al final de una lista, y en lugar de un terminal podemos usar la notación "**cualquiera**", indicando que si no se ha encontrado el símbolo de entrada actual hasta ese punto en la lista, debemos realizar esa acción sin importar lo que sea la entrada. Además, las entradas de error pueden sustituirse sin problemas por acciones de reducción, para una mayor uniformidad a lo largo de una fila. Los errores se detectarán más adelante, antes de un movimiento de desplazamiento.

Ejemplo 4.65: Considere la tabla de análisis sintáctico de la figura 4.37. En primer lugar, observe que las acciones para los estados 0, 4, 6 y 7 coinciden. Podemos representarlas todas mediante la siguiente lista:

SÍMBOLO	ACCIÓN
id	s5
(	s4
cualquiera	error

El estado 1 tiene una lista similar:

+	s6
$	acc
cualquiera	error

En el estado 2, podemos sustituir las entradas de error por r2, para que se realice la reducción mediante la producción 2 en cualquier entrada excepto $*$. Por ende, la lista para el estado 2 es:

*	s7
cualquiera	r2

El estado 3 tiene sólo entradas de error y r4. Podemos sustituir la primera por la segunda, de manera que la lista para el estado 3 consista sólo en el par (**cualquiera**, r4). Los estados 5, 10 y 11 pueden tratarse en forma similar. La lista para el estado 8 es:

+	s6
)	s11
cualquiera	error

y para el estado 9 es:

*	s7
)	s11
cualquiera	r1

□

También podemos codificar la tabla ir_A mediante una lista, pero aquí es más eficiente crear una lista de pares para cada no terminal A. Cada par en la lista para A es de la forma (*estadoActual*, *siguienteEstado*), lo cual indica que:

$$\text{ir\_A}[\textit{estadoActual}, A] = \textit{siguienteEstado}$$

Esta técnica es útil, ya que tiende a haber menos estados en cualquier columna de la tabla ir_A. La razón es que el ir_A en el no terminal A sólo puede ser un estado que pueda derivarse a partir de un conjunto de elementos en los que algunos elementos tengan a A justo a la izquierda de un punto. Ningún conjunto tiene elementos con X y Y justo a la izquierda de un punto si $X \neq Y$. Por ende, cada estado aparece como máximo en una columna ir_A.

Para una mayor reducción del espacio, hay que observar que las entradas de error en la tabla de ir_A nunca se consultan. Por lo tanto, podemos sustituir cada entrada de error por la entrada más común sin error en su columna. Esta entrada se convierte en la opción predeterminada; se representa en la lista para cada columna mediante un par con **cualquiera** en vez de *estadoActual*.

Ejemplo 4.66: Considere de nuevo la figura 4.37. La columna para F tiene la entrada 10 para el estado 7, y todas las demás entradas son 3 o error. Podemos sustituir error por 3 y crear, para la columna F, la siguiente lista:

ESTADOACTUAL	SIGUIENTEESTADO
7	10
cualquiera	3

De manera similar, una lista adecuada para la columna T es:

6	9
cualquiera	2

Para la columna E podemos elegir 1 o 8 como la opción predeterminada; son necesarias dos entradas en cualquier caso. Por ejemplo, podríamos crear para la columna E la siguiente lista:

4	8
cualquiera	1

□

El ahorro de espacio en estos pequeños ejemplos puede ser engañoso, ya que el número total de entradas en las listas creadas en este ejemplo y el anterior, junto con los apuntadores desde los estados hacia las listas de acción, y desde las no terminales hacia las listas de los siguientes estados, producen un ahorro de espacio mínimo, en comparación con la implementación de una matriz de la figura 4.37. En las gramáticas prácticas, el espacio necesario para la representa ción de la lista es, por lo general, menos del diez por ciento de lo necesario para la representación de la matriz. Los métodos de compresión de tablas para los autómatas finitos que vimos en la sección 3.9.8 pueden usarse también para representar las tablas de análisis sintáctico LR.

4.7.7 Ejercicios para la sección 4.7

Ejercicio 4.7.1: Construya los conjuntos de elementos

a) LR canónicos, y

b) LALR.

para la gramática $S \rightarrow S\ S + \mid S\ S * \mid a$ del ejercicio 4.2.1.

Ejercicio 4.7.2: Repita el ejercicio 4.7.1 para cada una de las gramáticas (aumentadas) del ejercicio 4.2.2(a)-(g).

! **Ejercicio 4.7.3:** Para la gramática del ejercicio 4.7.1, use el Algoritmo 4.63 para calcular la colección de conjuntos de elementos LALR, a partir de los corazones de los conjuntos de elementos LR(0).

! **Ejercicio 4.7.4:** Muestre que la siguiente gramática:

$$\begin{array}{lcl} S & \rightarrow & A\ a \mid b\ A\ c \mid d\ c \mid b\ d\ a \\ A & \rightarrow & d \end{array}$$

es LALR(1), pero no SLR(1).

! **Ejercicio 4.7.5:** Muestre que la siguiente gramática:

$$\begin{array}{lcl} S & \rightarrow & A\ a \mid b\ A\ c \mid B\ c \mid d\ B\ a \\ A & \rightarrow & d \\ B & \rightarrow & d \end{array}$$

es LR(1), pero no LALR(1).

4.8 Uso de gramáticas ambiguas

Es un hecho que ninguna gramática ambigua es LR y, por ende, no se encuentra en ninguna de las clases de gramáticas que hemos visto en las dos secciones anteriores. No obstante, ciertos tipos de gramáticas ambiguas son bastante útiles en la especificación e implementación de lenguajes. Para las construcciones de lenguajes como las expresiones, una gramática ambigua proporciona una especificación más corta y natural que cualquier gramática no ambigua equivalente. Otro uso de las gramáticas ambiguas es el de aislar las construcciones sintácticas que ocurren con frecuencia para la optimización de casos especiales. Con una gramática ambigua, podemos especificar las construcciones de casos especiales, agregando con cuidado nuevas producciones a la gramática.

Aunque las gramáticas que usamos no son ambiguas, en todos los casos especificamos reglas para eliminar la ambigüedad, las cuales sólo permiten un árbol de análisis sintáctico para cada enunciado. De esta forma, se eliminan las ambigüedades de la especificación general del lenguaje, y algunas veces es posible diseñar un analizador sintáctico LR que siga las mismas opciones para resolver las ambigüedades. Debemos enfatizar que las construcciones ambiguas deben utilizarse con medida y en un forma estrictamente controlada; de no ser así, no puede haber garantía en el lenguaje que reconozca un analizador sintáctico.

4.8.1 Precedencia y asociatividad para resolver conflictos

Considere la gramática ambigua (4.3) para las expresiones con los operadores + y $*$, que repetimos a continuación por conveniencia:

$$E \rightarrow E + E \mid E * E \mid (E) \mid \mathbf{id}$$

Esta gramática es ambigua, ya que no especifica la asociatividad ni la precedencia de los operadores + y $*$. La gramática sin ambigüedad (4.1), que incluye las producciones $E \rightarrow E + T$ y $T \rightarrow T * F$, genera el mismo lenguaje, pero otorga a + una precedencia menor que la de $*$, y hace que ambos operadores sean asociativos por la izquierda. Hay dos razones por las cuales podría ser más conveniente preferir el uso de la gramática ambigua. En primer lugar, como veremos más adelante, podemos cambiar con facilidad la asociatividad y la precedencia de los operadores + y $*$ sin perturbar las producciones de (4.3) o el número de estados en el analizador sintáctico resultante. En segundo lugar, el analizador sintáctico para la gramática sin ambigüedad invertirá una fracción considerable de su tiempo realizando reducciones mediante las producciones $E \rightarrow T$ y $T \rightarrow F$, cuya única función es hacer valer la asociatividad y la precedencia. El analizador sintáctico para la gramática sin ambigüedad (4.3) no desperdiciará tiempo realizando reducciones mediante estas producciones *simples* (producciones cuyo cuerpo consiste en un solo no terminal).

Los conjuntos de elementos LR(0) para la gramática de expresiones sin ambigüedad (4.3) aumentada por $E' \rightarrow E$ se muestran en la figura 4.48. Como la gramática (4.3) es ambigua, habrá conflictos de acciones de análisis sintáctico cuando tratemos de producir una tabla de análisis sintáctico LR a partir de los conjuntos de elementos. Los estados que corresponden a los conjuntos de elementos I_7 e I_8 generan estos conflictos. Suponga que utilizamos el método SLR para construir la tabla de acciones de análisis sintáctico. El conflicto generado por I_7 entre la reducción mediante $E \rightarrow E + E$ y el desplazamiento en + o $*$ no puede resolverse, ya que + y $*$ se encuentran en SIGUIENTE(E). Por lo tanto, se llamaría a ambas acciones en las entradas + y $*$. I_8 genera un conflicto similar, entre la reducción mediante $E \rightarrow E * E$ y el desplazamiento en las entradas + y $*$. De hecho, cada uno de nuestros métodos de construcción de tablas de análisis sintáctico LR generarán estos conflictos.

No obstante, estos problemas pueden resolverse mediante el uso de la información sobre la precedencia y la asociatividad para + y $*$. Considere la entrada **id** + **id** $*$ **id**, la cual hace que un analizador sintáctico basado en la figura 4.48 entre al estado 7 después de procesar **id** + **id**; de manera específica, el analizador sintáctico llega a la siguiente configuración:

PREFIJO	PILA	ENTRADA
$E + E$	0 1 4 7	$*$ **id** \$

Por conveniencia, los símbolos que corresponden a los estados 1, 4 y 7 también se muestran bajo PREFIJO.

Si $*$ tiene precedencia sobre +, sabemos que el analizador sintáctico debería desplazar a $*$ hacia la pila, preparándose para reducir el $*$ y sus símbolos **id** circundantes a una expresión. El analizador sintáctico SLR de la figura 4.37 realizó esta elección, con base en una gramática sin ambigüedad para el mismo lenguaje. Por otra parte, si + tiene precedencia sobre $*$, sabemos que el analizador sintáctico debería reducir $E + E$ a E. Por lo tanto, la precedencia relativa

I_0: $E' \to \cdot E$
$E \to \cdot E + E$
$E \to \cdot E * E$
$E \to \cdot(E)$
$E \to \cdot\mathbf{id}$

I_1: $E' \to E\cdot$
$E \to E\cdot + E$
$E \to E\cdot * E$

I_2: $E \to (\cdot E)$
$E \to \cdot E + E$
$E \to \cdot E * E$
$E \to \cdot(E)$
$E \to \cdot\mathbf{id}$

I_3: $E \to \mathbf{id}\cdot$

I_4: $E \to E + \cdot E$
$E \to \cdot E + E$
$E \to \cdot E * E$
$E \to \cdot(E)$
$E \to \cdot\mathbf{id}$

I_5: $E \to E * \cdot E$
$E \to \cdot E + E$
$E \to \cdot E * E$
$E \to \cdot(E)$
$E \to \cdot\mathbf{id}$

I_6: $E \to (E\cdot)$
$E \to E\cdot + E$
$E \to E\cdot * E$

I_7: $E \to E + E\cdot$
$E \to E\cdot + E$
$E \to E\cdot * E$

I_8: $E \to E * E\cdot$
$E \to E\cdot + E$
$E \to E\cdot * E$

I_9: $E \to (E)\cdot$

Figura 4.48: Conjuntos de elementos LR(0) para una gramática de expresiones aumentada

de + seguido de $*$ determina en forma única la manera en que debería resolverse el conflicto de acciones de análisis sintáctico entre la reducción $E \to E + E$ y el desplazamiento sobre $*$ en el estado 7.

Si la entrada hubiera sido **id** + **id** + **id**, el analizador sintáctico llegaría de todas formas a una configuración en la cual tendría la pila 0 1 4 7 después de procesar la entrada **id** + **id**. En la entrada + hay de nuevo un conflicto de desplazamiento/reducción en el estado 7. Sin embargo, ahora la asociatividad del operador + determina cómo debe resolverse este conflicto. Si + es asociativo a la izquierda, la acción correcta es reducir mediante $E \to E + E$. Es decir, los símbolos **id** que rodean el primer + deben agruparse primero. De nuevo, esta elección coincide con lo que haría el analizador sintáctico SLR para la gramática sin ambigüedad.

En resumen, si asumimos que + es asociativo por la izquierda, la acción del estado 7 en la entrada + debería ser reducir mediante $E \to E + E$, y suponiendo que $*$ tiene precedencia sobre +, la acción del estado 7 en la entrada $*$ sería desplazar. De manera similar, suponiendo que $*$ sea asociativo por la izquierda y tenga precedencia sobre +, podemos argumentar que el estado 8, que puede aparecer en la parte superior de la pila sólo cuando $E * E$ son los tres símbolos gramaticales de la parte superior, debería tener la acción de reducir $E \to E * E$ en las entradas + y $*$. En el caso de la entrada +, la razón es que $*$ tiene precedencia sobre +, mientras que en el caso de la entrada $*$, el fundamento es que $*$ es asociativo por la izquierda.

Si procedemos de esta forma, obtendremos la tabla de análisis sintáctico LR que se muestra en la figura 4.49. Las producciones de la 1 a la 4 son $E \rightarrow E + E$, $E \rightarrow E * E$, $\rightarrow (E)$ y $E \rightarrow$ **id**, respectivamente. Es interesante que una tabla de acciones de análisis sintáctico similar se produzca eliminando las reducciones mediante las producciones simples $E \rightarrow T$ y $T \rightarrow F$ a partir de la tabla SLR para la gramática de expresiones sin ambigüedad (4.1) que se muestra en la figura 4.37. Las gramáticas ambiguas como la que se usa para las expresiones pueden manejarse en una forma similar, en el contexto de los análisis sintácticos LALR y LR canónico.

Estado	Acción						ir_a
	id	+	*	(	)	$	*E*
0	s3			s2			1
1		s4	s5			acc	
2	s3			s2			6
3		r4	r4		r4	r4	
4	s3			s2			7
5	s3			s2			8
6		s4	s5		s9		
7		r1	s5		r1	r1	
8		r2	r2		r2	r2	
9		r3	r3		r3	r3	

Figura 4.49: Tabla de análisis sintáctico para la gramática (4.3)

4.8.2 La ambigüedad del "else colgante"

Considere de nuevo la siguiente gramática para las instrucciones condicionales:

$$\begin{array}{rcl} \textit{instr} & \rightarrow & \textbf{if } \textit{expr} \textbf{ then } \textit{instr} \textbf{ else } \textit{instr} \\ & | & \textbf{if } \textit{expr} \textbf{ then } \textit{instr} \\ & | & \textbf{otras} \end{array}$$

Como vimos en la sección 4.3.2, esta gramática no tiene ambigüedades, ya que no resuelve la ambigüedad del else colgante. Para simplificar la discusión, vamos a considerar una abstracción de esta gramática, en donde i representa a **if** *expr* **then**, e representa a **else**, y a representa a "todas las demás producciones". De esta forma podemos escribir la gramática, con la producción aumentada $S' \rightarrow S$, como

$$\begin{array}{rcl} S' & \rightarrow & S \\ S & \rightarrow & i\ S\ e\ S \mid i\ S \mid a \end{array} \tag{4.67}$$

Los conjuntos de elementos LR(0) para la gramática (4.67) se muestran en la figura 4.50. La ambigüedad en (4.67) produce un conflicto de desplazamiento/reducción en I_4. Ahí, $S \rightarrow iS \cdot eS$ llama a un desplazamiento de e y, como SIGUIENTE$(S) = \{e, \$\}$, el elemento $S \rightarrow iS \cdot$ llama a la reducción mediante $S \rightarrow iS$ en la entrada e.

I_0: $S' \to \cdot S$
$S \to \cdot iSeS$
$S \to \cdot iS$
$S \to \cdot a$

I_1: $S' \to S\cdot$

I_2: $S \to i\cdot SeS$
$S \to i\cdot S$
$S \to \cdot iSeS$
$S \to \cdot iS$
$S \to \cdot a$

I_3: $S \to a\cdot$

I_4: $S \to iS\cdot eS$

I_5: $S \to iSe\cdot S$
$S \to \cdot iSeS$
$S \to \cdot iS$
$S \to \cdot a$

I_6: $S \to iSeS\cdot$

Figura 4.50: Estados LR(0) para la gramática aumentada (4.67)

Traduciendo esto de vuelta a la terminología **if-then-else**, si tenemos a:

if *expr* **then** *instr*

en la pila y a **else** como el primer símbolo de entrada, ¿debemos desplazar el **else** hacia la pila (es decir, desplazar a e) o reducir **if** *expr* **then** *instr* (es decir, reducir mediante $S \to iS$)? La respuesta es que debemos desplazar el **else**, ya que está "asociado" con el **then** anterior. En la terminología de la gramática (4.67), la e en la entrada, que representa a **else**, sólo puede formar parte del cuerpo que empieza con la iS que está ahora en la parte superior de la pila. Si lo que sigue después de e en la entrada no puede analizarse como una S, para completar el cuerpo $iSeS$, entonces podemos demostrar que no hay otro análisis sintáctico posible.

Concluimos que el conflicto de desplazamiento/reducción en I_4 debe resolverse a favor del desplazamiento en la entrada e. La tabla de análisis sintáctico SLR que se construyó a partir de los conjuntos de elementos de la figura 4.48, que utiliza esta resolución del conflicto de acciones de análisis sintáctico en I_4 con la entrada e, se muestra en la figura 4.51. Las producciones de la 1 a la 3 son $S \to iSeS$, $S \to iS$ y $S \to a$, respectivamente.

Estado	Acción				ir_A
	i	e	a	\$	S
0	s2		s3		1
1				acc	
2	s2		s3		4
3		r3		r3	
4		s5		r2	
5	s2		s3		6
6		r1		r1	

Figura 4.51: Tabla de análisis sintáctico LR para la gramática del "else colgante"

Por ejemplo, en la entrada *iiaea*, el analizador sintáctico realiza los movimientos que se muestran en la figura 4.52, correspondientes a la resolución correcta del "else colgante". En la línea (5), el estado 4 selecciona la acción de desplazamiento en la entrada *e*, mientras que en la línea (9), el estado 4 llama a la reducción mediante $S \rightarrow iS$ en la entrada \$.

	Pila	Símbolos	Entrada	Acción
(1)	0		$iiaea\$$	desplazar
(2)	0 2	i	$iaea\$$	desplazar
(3)	0 2 2	ii	$aea\$$	desplazar
(4)	0 2 2 3	iia	$ea\$$	desplazar
(5)	0 2 2 4	iiS	$ea\$$	reducir $S \rightarrow a$
(6)	0 2 2 4 5	$iiSe$	$a\$$	desplazar
(7)	0 2 2 4 5 3	$iiSea$	$\$$	reducir $S \rightarrow a$
(8)	0 2 2 4 5 6	$iiSeS$	$\$$	reducir $S \rightarrow iSeS$
(9)	0 2 4	iS	$\$$	reducir $S \rightarrow iS$
(10)	0 1	S	$\$$	aceptar

Figura 4.52: Acciones de análisis sintáctico con la entrada *iiaea*

Con el fin de comparar, si no podemos usar una gramática ambigua para especificar instrucciones condicionales, entonces tendríamos que usar una gramática más robusta a lo largo de las líneas del ejemplo 4.16.

4.8.3 Recuperación de errores en el análisis sintáctico LR

Un analizador sintáctico LR detectará un error al consultar la tabla de acciones de análisis sintáctico y encontrar una entrada de error. Los errores nunca se detectan al consultar la tabla de ir_A. Un analizador sintáctico LR anunciará un error tan pronto como no haya una continuación válida para la porción de la entrada que se ha explorado hasta ese momento. Un analizador sintáctico LR canónico no realizara ni siquiera una sola reducción antes de anunciar un error. Los analizadores sintácticos SLR y LALR pueden realizar varias reducciones antes de anunciar un error, pero nunca desplazarán un símbolo de entrada erróneo hacia la pila.

En el análisis sintáctico LR, podemos implementar la recuperación de errores en modo de pánico de la siguiente manera. Exploramos la pila en forma descendente hasta encontrar un estado s con un ir_A en un no terminal A específico. Después, se descartan cero o más símbolos de entrada hasta encontrar un símbolo a que pueda seguir a A de manera legítima. A continuación, el analizador sintáctico mete el estado ir_A(s, A) en la pila y continúa con el análisis sintáctico normal. Podría haber más de una opción para el no terminal A. Por lo general, éstos serían no terminales que representen las piezas principales del programa, como una expresión, una instrucción o un bloque. Por ejemplo, si A es el no terminal *instr*, a podría ser un punto y coma o }, lo cual marca el final de una secuencia de instrucciones.

Este método de recuperación de errores trata de eliminar la frase que contiene el error sintáctico. El analizador sintáctico determina que una cadena que puede derivarse de A contiene un error. Parte de esa cadena ya se ha procesado, y el resultado de este procesamiento es una

secuencia de estados en la parte superior de la pila. El resto de la cadena sigue en la entrada, y el analizador sintáctico trata de omitir el resto de esta cadena buscando un símbolo en la entrada que pueda seguir de manera legítima a A. Al eliminar estados de la pila, el analizador sintáctico simula que ha encontrado una instancia de A y continúa con el análisis sintáctico normal.

Para implementar la recuperación a nivel de frase, examinamos cada entrada de error en la tabla de análisis sintáctico LR y decidimos, en base al uso del lenguaje, el error más probable del programador que pudiera ocasionar ese error. Después podemos construir un procedimiento de recuperación de errores apropiado; se supone que la parte superior de la pila y los primeros símbolos de entrada se modificarían de una forma que se considera como apropiada para cada entrada de error.

Al diseñar rutinas de manejo de errores específicas para un analizador sintáctico LR, podemos rellenar cada entrada en blanco en el campo de acción con un apuntador a una rutina de error que tome la acción apropiada, seleccionada por el diseñador del compilador. Las acciones pueden incluir la inserción o eliminación de símbolos de la pila o de la entrada (o de ambas), o la alteración y transposición de los símbolos de entrada. Debemos realizar nuestras elecciones de tal forma que el analizador sintáctico LR no entre en un ciclo infinito. Una estrategia segura asegurará que por lo menos se elimine o se desplace un símbolo de entrada en un momento dado, o que la pila se reduzca si hemos llegado al final de la entrada. Debemos evitar sacar un estado de la pila que cubra un no terminal, ya que esta modificación elimina de la pila una construcción que ya se haya analizado con éxito.

Ejemplo 4.68: Considere de nuevo la siguiente gramática de expresiones:

$$E \rightarrow E + E \mid E * E \mid (E) \mid \mathbf{id}$$

La figura 4.53 muestra la tabla de análisis sintáctico LR de la figura 4.49 para esta gramática, modificada para la detección y recuperación de errores. Hemos modificado cada estado que llama a una reducción específica en ciertos símbolos de entrada, mediante la sustitución de las entradas de error en ese estado por la reducción. Este cambio tiene el efecto de posponer la detección de errores hasta que se realicen una o más reducciones, pero el error seguirá atrapándose antes de que se realice cualquier desplazamiento. Las entradas restantes en blanco de la figura 4.49 se han sustituido por llamadas a las rutinas de error.

Las rutinas de error son las siguientes:

e1: Esta rutina se llama desde los estados 0, 2, 4 y 5, y todos ellos esperan el principio de un operando, ya sea un **id** o un paréntesis izquierdo. En vez de ello, se encontró +, $*$ o el final de la entrada.

> meter el estado 3 (el ir_A de los estados 0, 2, 4 y 5 en **id**);
> emitir el diagnóstico "falta operando".

e2: Se llama desde los estados 0, 1, 2, 4 y 5 al encontrar un paréntesis derecho.

> eliminar el paréntesis derecho de la entrada;
> emitir el diagnóstico "paréntesis derecho desbalanceado".

Estado	Acción						ir_a
	id	+	*	(	)	$	*E*
0	s3	e1	e1	s2	e2	e1	1
1	e3	s4	s5	e3	e2	acc	
2	s3	e1	e1	s2	e2	e1	6
3	r4	r4	r4	r4	r4	r4	
4	s3	e1	e1	s2	e2	e1	7
5	s3	e1	e1	s2	e2	e1	8
6	e3	s4	s5	e3	s9	e4	
7	r1	r1	s5	r1	r1	r1	
8	r2	r2	r2	r2	r2	r2	
9	r3	r3	r3	r3	r3	r3	

Figura 4.53: Tabla de análisis sintáctico LR con rutinas de error

e3: Se llama desde los estados 1 o 6 cuando se espera un operador y se encuentra un **id** o paréntesis derecho.

> meter el estado 4 (correspondiente al símbolo +) en la pila;
> emitir el diagnóstico "falta un operador".

e4: Se llama desde el estado 6 cuando se encuentra el final de la entrada.

> meter el estado 9 (para un paréntesis derecho) a la pila;
> emitir el diagnóstico "falta paréntesis derecho".

En la entrada errónea **id** +), la secuencia de configuraciones que introduce el analizador sintáctico se muestra en la figura 4.54. □

4.8.4 Ejercicios para la sección 4.8

! **Ejercicio 4.8.1:** La siguiente es una gramática ambigua para las expresiones con n operadores binarios infijo, con n niveles distintos de precedencia:

$$E \rightarrow E\,\theta_1\,E \mid E\,\theta_2\,E \mid \cdots E\,\theta_n\,E \mid (\,E\,) \mid \mathbf{id}$$

a) Como una función de n, ¿cuáles son los conjuntos de elementos SLR?

b) ¿Cómo resolvería los conflictos en los elementos SLR, de manera que todos los operadores sean asociativos a la izquierda, y que θ_1 tenga precedencia sobre θ_2, que tiene precedencia sobre θ_3, y así sucesivamente?

c) Muestre la tabla de análisis sintáctico SLR que resulta de sus decisiones en la parte (b).

Pila	Símbolos	Entrada	Acción
0		**id** +) $	
0 3	**id**	+) $	
0 1	E	+) $	
0 1 4	$E+$	) $	"paréntesis derecho desbalanceado"
0 1 4	$E+$	$	e2 elimina el paréntesis derecho "falta un operando"
0 1 4 3	$E+$ **id**	$	e1 mete el estado 3
0 1 4 7	$E+$	$	en la pila
0 1	$E+$	$	

Figura 4.54: Movimientos de análisis sintáctico y recuperación de errores realizados por un analizador sintáctico LR

d) Repita las partes (a) y (c) para la gramática sin ambigüedad, la cual define el mismo conjunto de expresiones, como se muestra en la figura 4.55.

e) ¿Cómo se comparan los conteos del número de conjuntos de elementos y los tamaños de las tablas para las dos gramáticas (ambigua y sin ambigüedad)? ¿Qué nos dice esa comparación acerca del uso de las gramáticas de expresiones ambiguas?

$$
\begin{array}{lcl}
E_1 & \rightarrow & E_1\ \theta\ E_2 \mid E_2 \\
E_2 & \rightarrow & E_2\ \theta\ E_3 \mid E_3 \\
 & \cdots & \\
E_n & \rightarrow & E_n\ \theta\ E_{n+1} \mid E_{n+1} \\
E_{n+1} & \rightarrow & (\ E_1\ ) \mid \textbf{id}
\end{array}
$$

Figura 4.55: Gramática sin ambigüedad para n operadores

! Ejercicio 4.8.2: En la figura 4.56 hay una gramática para ciertas instrucciones, similar a la que vimos en el ejercicio 4.4.12. De nuevo, e y s son terminales que representan expresiones condicionales y "otras instrucciones", respectivamente.

a) Construya una tabla de análisis sintáctico LR para esta gramática, resolviendo los conflictos de la manera usual para el problema del else colgante.

b) Implemente la corrección de errores, llenando las entradas en blanco en la tabla de análisis sintáctico con acciones de reducción adicionales, o rutinas de recuperación de errores adecuadas.

c) Muestre el comportamiento de su analizador sintáctico con las siguientes entradas:

(i) **if** e **then** s ; **if** e **then** s **end**
(ii) **while** e **do begin** s ; **if** e **then** s ; **end**

$$
\begin{array}{lcl}
\mathit{instr} & \rightarrow & \textbf{if } e \textbf{ then } \mathit{instr} \\
& | & \textbf{if } e \textbf{ then } \mathit{instr} \textbf{ else } \mathit{instr} \\
& | & \textbf{while } e \textbf{ do } \mathit{instr} \\
& | & \textbf{begin } \mathit{lista} \textbf{ end} \\
& | & s \\
\mathit{lista} & \rightarrow & \mathit{lista} \ ; \ \mathit{instr} \\
& | & \mathit{instr}
\end{array}
$$

Figura 4.56: Una gramática para ciertos tipos de instrucciones

4.9 Generadores de analizadores sintácticos

En esta sección veremos cómo puede usarse un generador de analizadores sintácticos para facilitar la construcción del front-end de usuario de un compilador. Utilizaremos el generador de analizadores sintácticos LALR de nombre `Yacc` como la base de nuestra explicación, ya que implementa muchos de los conceptos que vimos en las dos secciones anteriores, y se emplea mucho. `Yacc` significa "yet another compiler-compiler" (otro compilador-de compiladores más), lo cual refleja la popularidad de los generadores de analizadores sintácticos a principios de la década de 1970, cuando S. C. Johnson creó la primera versión de `Yacc`. Este generador está disponible en forma de comando en el sistema en UNIX, y se ha utilizado para ayudar a implementar muchos compiladores de producción.

4.9.1 El generador de analizadores sintácticos `Yacc`

Puede construirse un traductor mediante el uso de `Yacc` de la forma que se ilustra en la figura 4.57. En primer lugar se prepara un archivo, por decir `traducir.y`, el cual contiene una especificación de `Yacc` del traductor. El siguiente comando del sistema UNIX:

```
yacc traducir.y
```

transforma el archivo `traducir.y` en un programa en C llamado `y.tab.c`, usando el método LALR descrito en el algoritmo 4.63. El programa `y.tab.c` es una representación de un analizador sintáctico LALR escrito en C, junto con otras rutinas en C que el usuario puede haber preparado. La tabla de análisis sintáctico LR se compacta según lo descrito en la sección 4.7. Al compilar `y.tab.c` junto con la biblioteca `ly` que contiene el programa de análisis sintáctico LR mediante el uso del comando:

```
cc y.tab.c -ly
```

obtenemos el programa objeto `a.out` deseado, el cual realiza la traducción especificada por el programa original en `Yacc`.[7] Si se necesitan otros procedimientos, pueden compilarse o cargarse con `y.tab.c`, de igual forma que con cualquier programa en C.

Un programa fuente en `Yacc` tiene tres partes:

[7] El nombre `ly` es dependiente del sistema.

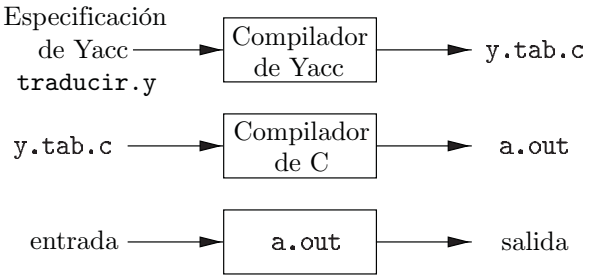

Figura 4.57: Creación de un traductor de entrada/salida con `Yacc`

```
declaraciones
%%
reglas de traducción
%%
soporte de las rutinas en C
```

Ejemplo 4.69: Para ilustrar cómo preparar un programa fuente en `Yacc`, vamos a construir una calculadora de escritorio simple que lee una expresión aritmética, la evalúa e imprime su valor numérico. Vamos a construir la calculadora de escritorio empezando con la siguiente gramática para las expresiones aritméticas:

$$\begin{array}{lcl} E & \rightarrow & E + T \mid T \\ T & \rightarrow & T * F \mid F \\ R & \rightarrow & (E) \mid \textbf{digit} \end{array}$$

El token **digito** es un solo dígito entre 0 y 9. En la figura 4.58 se muestra un programa de calculadora de escritorio en `Yacc`, derivado a partir de esta gramática. □

La parte de las declaraciones

Hay dos secciones en la parte de las declaraciones de un programa en `Yacc`; ambas son opcionales. En la primera sección, colocamos las declaraciones ordinarias en C, delimitadas mediante `%{` y `%}`. Aquí colocamos las declaraciones de cualquier valor temporal usado por las reglas de traducción o los procedimientos de las secciones segunda y tercera. En la figura 4.58, esta sección contiene sólo la siguiente instrucción de inclusión:

```
#include <ctype.h>
```

la cual ocasiona que el preprocesador de C incluya el archivo de encabezado estándar `<ctype.h>`, el cual contiene el predicado `isdigit`.

Además, en la parte de las declaraciones se encuentran las declaraciones de los tokens de gramática. En la figura 4.58, la instrucción

```
%token DIGITO
```

```
%{
#include <ctype.h>
%}

%token DIGITO

%%
linea   :  expr '\n'         { printf("%d\n", $1); }
        ;
expr    :  expr '+' term     { $$ = $1 + $3; }
        |  term
        ;
term    :  term '*' factor   { $$ = $1 * $3; }
        |  factor
        ;
factor  :  '(' expr ')'      { $$ = $2; }
        |  DIGITO
        ;
%%
yylex() {
    int c;
    c = getchar();
    if (isdigit(c)) {
        yylval = c-'0';
        return DIGITO;
    }
    return c;
}
```

Figura 4.58: Especificación de `Yacc` de una calculadora de escritorio simple

declara a `DIGITO` como un token. Los tokens declarados en esta sección pueden usarse en las partes segunda y tercera de la especificación de `Yacc`. Si se utiliza `Lex` para crear el analizador léxico que pasa el token al analizador sintáctico `Yacc`, entonces estas declaraciones de tokens también se vuelven disponibles para el analizador generado por `Lex`, como vimos en la sección 3.5.2.

La parte de las reglas de traducción

En la parte de la especificación de `Yacc` después del primer par de `%%`, colocamos las reglas de traducción. Cada regla consiste en una producción gramatical y la acción semántica asociada. Un conjunto de producciones que hemos estado escribiendo como:

$$\langle \text{encabezado} \rangle \quad \rightarrow \quad \langle \text{cuerpo} \rangle_1 \mid \langle \text{cuerpo} \rangle_2 \mid \cdots \mid \langle \text{cuerpo} \rangle_n$$

podría escribirse en `Yacc` de la siguiente manera:

⟨encabezado⟩ : ⟨cuerpo⟩$_1$ { ⟨acción semántica⟩$_1$ }
| ⟨cuerpo⟩$_2$ { ⟨acción semántica⟩$_2$ }
…
| ⟨cuerpo⟩$_n$ { ⟨acción semántica⟩$_n$ }
;

En una producción de `Yacc`, las cadenas sin comillas de letras y dígitos que no se declaren como tokens se consideran como no terminales. Un solo carácter entre comillas, por ejemplo `'c'`, se considera como el símbolo terminal `c`, así como el código entero para el token representado por ese carácter (es decir, `Lex` devolvería el código de carácter para `'c'` al analizador sintáctico, como un entero). Los cuerpos alternativos pueden separarse mediante una barra vertical; además se coloca un punto y coma después de cada encabezado con sus alternativas y sus acciones semánticas. El primer encabezado se considera como el símbolo inicial.

Una acción semántica de `Yacc` es una secuencia de instrucciones en C. En una acción semántica, el símbolo `$$` se refiere al valor del atributo asociado con el no terminal del encabezado, mientras que `$`i se refiere al valor asociado con el i-ésimo símbolo gramatical (terminal o no terminal) del cuerpo. La acción semántica se realiza cada vez que reducimos mediante la producción asociada, por lo que normalmente la acción semántica calcula un valor para `$$` en términos de los `$`i's. En la especificación de `Yacc`, hemos escrito las dos producciones E siguientes:

$$E \rightarrow E + T \mid T$$

y sus acciones semánticas asociadas como:

```
expr : expr '+' term    { $$ = $1 + $3; }
     | term
     ;
```

Observe que el no terminal `term` en la primera producción es el tercer símbolo gramatical del cuerpo, mientras que + es el segundo. La acción semántica asociada con la primera producción agrega el valor de la `expr` y la `term` del cuerpo, y asigna el resultado como el valor para el no terminal `expr` del encabezado. Hemos omitido del todo la acción semántica para la segunda producción, ya que copiar el valor es la acción predeterminada para las producciones con un solo símbolo gramatical en el cuerpo. En general, `{ $$ = $1; }` es la acción semántica predeterminada.

Observe que hemos agregado una nueva producción inicial:

```
linea : expr '\n'    { printf("%d\n", $1); }
```

a la especificación de `Yacc`. Esta producción indica que una entrada para la calculadora de escritorio debe ser una expresión seguida de un carácter de nueva línea. La acción semántica asociada con esta producción imprime el valor decimal de la expresión que va seguida de un carácter de nueva línea.

La parte de las rutinas de soporte en C

La tercera parte de una especificación de `Yacc` consiste en las rutinas de soporte en C. Debe proporcionarse un analizador léxico mediante el nombre `yylex()`. La elección común es usar `Lex` para producir `yylex()`; vea la sección 4.9.3. Pueden agregarse otros procedimientos como las rutinas de recuperación de errores, según sea necesario.

El analizador léxico `yylex()` produce tokens que consisten en un nombre de token y su valor de atributo asociado. Si se devuelve el nombre de un token como `DIGITO`, el nombre del token debe declararse en la primera sección de la especificación de `Yacc`. El valor del atributo asociado con un token se comunica al analizador sintáctico, a través de una variable `yylval` definida por `Yacc`.

El analizador léxico en la figura 4.58 es bastante burdo. Lee un carácter de entrada a la vez, usando la función de C `getchar()`. Si el carácter es un dígito, el valor del dígito se almacena en la variable `yylval` y se devuelve el nombre de token `DIGITO`. En cualquier otro caso, se devuelve el mismo carácter como el nombre de token.

4.9.2 Uso de Yacc con gramáticas ambiguas

Ahora vamos a modificar la especificación de `Yacc`, de tal forma que la calculadora de escritorio resultante sea más útil. En primer lugar, vamos a permitir que la calculadora de escritorio evalúe una secuencia de expresiones, de una a una línea. También vamos a permitir líneas en blanco entre las expresiones. Para ello, cambiaremos la primer regla a:

```
lineas : lineas expr '\n'    { printf("%g\n", $2); }
       | lineas '\n'
       | /* vacia */
       ;
```

En `Yacc`, una alternativa vacía, como lo es la tercera línea, denota a ϵ.

En segundo lugar, debemos agrandar la clase de expresiones para incluir números en vez de dígitos individuales, y para incluir los operadores aritméticos $+$, $-$, (tanto binarios como unarios), $*$ y $/$. La manera más sencilla de especificar esta clase de expresiones es utilizar la siguiente gramática ambigua:

$$E \rightarrow E + E \mid E - E \mid E * E \mid E / E \mid - E \mid \textbf{numero}$$

La especificación resultante de `Yacc` se muestra en la figura 4.59.

Como la gramática en la especificación de `Yacc` en la figura 4.59 es ambigua, el algoritmo LALR generará conflictos de acciones de análisis sintáctico. `Yacc` reporta el número de conflictos de acciones de análisis sintáctico que se generan. Podemos obtener una descripción de los conjuntos de elementos y los conflictos de acciones de análisis sintáctico si invocamos a `Yacc` con una opción `-v`. Esta opción genera un archivo adicional `y.output`, el cual contiene los corazones de los conjuntos de elementos encontrados para la gramática, una descripción de los conflictos de acciones de análisis sintáctico generados por el algoritmo LALR, y una representación legible de la tabla de análisis sintáctico LR que muestra cómo se resolvieron los conflictos de acciones de análisis sintáctico. Cada vez que `Yacc` reporta que ha encontrado

```
%{
#include <ctype.h>
#include <stdio.h>
#define YYSTYPE double   /* tipo double para la pila de Yacc */
%}
%token NUMERO

%left '+', '-'
%left '*', '/'
%right UMENOS
%%

lines  : lines expr '\n'   { printf("%g\n", $2); }
       | lines '\n'
       | /* vacia */

expr   : expr '+' expr     { $$ = $1 + $3; }
       | expr '-' expr     { $$ = $1 - $3; }
       | expr '*' expr     { $$ = $1 * $3; }
       | expr '/' expr     { $$ = $1 / $3; }
       | '(' expr ')'      { $$ = $2; }
       | '-' expr %prec UMENOS { $$ = - $2; }
       | NUMERO
       ;
%%
yylex() {
    int c;
    while ( ( c = getchar() ) ) == ' ' );
    if ( ( c == '.') || (isdigit(c)) ) {
        unget(c, stdin);
        scanf( "%lf", &yyval);
        return NUMERO;
    }
    return c;
}
```

Figura 4.59: Especificación de `Yacc` para una calculadora de escritorio más avanzada

conflictos de acciones de análisis sintáctico, es conveniente crear y consultar el archivo `y.output` para ver por qué se generaron los conflictos de acciones de análisis sintáctico y si se resolvieron en forma correcta.

A menos que se indique lo contrario, `Yacc` resolverá todos los conflictos de las acciones de análisis sintáctico mediante las siguientes dos reglas:

1. Un conflicto de reducción/reducción se resuelve eligiendo la producción en conflicto que se presente primero en la especificación de `Yacc`.
2. Un conflicto de desplazamiento/reducción se resuelve a favor del desplazamiento. Esta regla resuelve en forma correcta el conflicto de desplazamiento/reducción ocasionado por la ambigüedad del else colgante.

Como estas reglas predeterminadas no siempre pueden ser lo que desea el escritor de compiladores, `Yacc` proporciona un mecanismo general para resolver los conflictos de desplazamiento/reducción. En la porción de las declaraciones, podemos asignar precedencias y asociatividades a las terminales. La siguiente declaración:

```
%left '+' '-'
```

hace que `+` y `-` sean de la misma precedencia y asociativos a la izquierda. Podemos declarar un operador como asociativo a la derecha si escribimos lo siguiente:

```
%right '^'
```

y podemos forzar a un operador para que sea un operador binario sin asociatividad (es decir, no pueden combinarse dos ocurrencias del operador de ninguna manera) escribiendo lo siguiente:

```
%nonassoc '<'
```

Los tokens reciben las precedencias en el orden en el que aparecen en la parte de las declaraciones, en donde la menor precedencia va primero. Los tokens en la misma declaración tienen la misma precedencia. Así, la declaración

```
%right UMENOS
```

en la figura 4.59 proporciona al token `UMENOS` un nivel de precedencia mayor que el de las cinco terminales anteriores.

`Yacc` resuelve los conflictos de desplazamiento/reducción adjuntando una precedencia y una asociatividad a cada una de las producciones involucradas en un conflicto, así como también a cada terminal involucrada en un conflicto. Si debe elegir entre desplazar el símbolo de entrada a y reducir mediante la producción $A \rightarrow \alpha$, `Yacc` reduce si la precedencia de la producción es mayor que la de a, o si las precedencias son iguales y la asociatividad de la producción es `left`. En cualquier otro caso, el desplazamiento es la acción elegida.

Por lo general, la precedencia de una producción se considera igual a el de su terminal por la derecha. Ésta es la decisión sensata en la mayoría de los casos. Por ejemplo, dadas las siguientes producciones:

$$E \rightarrow E + E \mid E + E$$

sería preferible reducir mediante $E \rightarrow E$+E con el símbolo de anticipación +, ya que el + en el cuerpo tiene la misma precedencia que el símbolo de anticipación, pero es asociativo a la izquierda. Con el símbolo de anticipación *, sería más preferible desplazar, ya que éste tiene una precedencia mayor que la del + en la producción.

En esas situaciones en las que el terminal por la derecha no proporciona la precedencia apropiada a una producción, podemos forzar el uso de una precedencia si adjuntamos a una producción la siguiente etiqueta:

%prec ⟨terminal⟩

La precedencia y la asociatividad de la producción serán entonces iguales que la del terminal, que se supone está definida en la sección de declaraciones. Yacc no reporta los conflictos de desplazamiento/reducción que se resuelven usando este mecanismo de precedencia y asociatividad.

Este "terminal" podría ser un receptáculo como UMENOS en la figura 4.59: el analizador léxico no devuelve este terminal, sino que está declarado con el único fin de definir una precedencia para una producción. En la figura 4.59, la declaración

```
%right UMENOS
```

asigna al token UMENOS una precedencia mayor que la de $*$ y /. En la parte de las reglas de traducción, la etiqueta:

```
%prec UMENOS
```

al final de la producción

```
expr  : '-' expr
```

hace que el operador de resta unario en esta producción tenga una menor precedencia que cualquier otro operador.

4.9.3 Creación de analizadores léxicos de Yacc con Lex

Lex se diseñó para producir analizadores léxicos que pudieran utilizarse con Yacc. La biblioteca ll de Lex proporciona un programa controlador llamado yylex(), el nombre que Yacc requiere para su analizador léxico. Si se utiliza Lex para producir el analizador léxico, sustituimos la rutina yylex() en la tercera parte de la especificación de Yacc con la siguiente instrucción:

```
#include "lex.yy.c"
```

y hacemos que cada acción de Lex devuelva un terminal conocido a Yacc. Al usar la instrucción #include "lex.yy.c", el programa yylex tiene acceso a los nombres de Yacc para los tokens, ya que el archivo de salida de Lex se compila como parte del archivo de salida y.tab.c de Yacc.

En el sistema UNIX, si la especificación de Lex está en el archivo primero.l y la especificación de Yacc en segundo.y, podemos escribir lo siguiente:

```
lex primero.l
yacc segundo.y
cc y.tab.c -ly -ll
```

para obtener el traductor deseado.

La especificación de `Lex` en la figura 4.60 puede usarse en vez del analizador léxico de la figura 4.59. El último patrón, que significa "cualquier carácter", debe escribirse como `\nl.` ya que el punto en Lex coincide con cualquier carácter, excepto el de nueva línea.

```
numero    [0-9]+\e.?|[0-9]*\e.[0-9]+
%%
[ ]       { /* omitir espacios en blanco */ }
{numero}  { sscanf(yytext, "%lf", &yylval);
                return NUMERO; }
\nl.     { return yytext[0]; }
```

Figura 4.60: Especificación de Lex para `yylex()` en la figura 4.59

4.9.4 Recuperación de errores en `Yacc`

En `Yacc`, la recuperación de errores utiliza una forma de producciones de error. En primer lugar, el usuario decide qué no terminales "importantes" tendrán la recuperación de errores asociado con ellas. Las elecciones típicas son cierto subconjunto de los no terminales que generan expresiones, instrucciones, bloques y funciones. Después el usuario agrega a las producciones de error gramaticales de la forma $A \rightarrow$ **error** α, en donde A es un no terminal importante y α es una cadena de símbolos gramaticales, tal vez la cadena vacía; **error** es una palabra reservada de `Yacc`. `Yacc` generará un analizador sintáctico a partir de dicha especificación, tratando a las producciones de error como producciones ordinarias.

No obstante, cuando el analizador sintáctico generado por `Yacc` encuentra un error, trata a los estados cuyos conjuntos de elementos contienen producciones de error de una manera especial. Al encontrar un error, `Yacc` saca símbolos de su pila hasta que encuentra el estado en la parte superior de su pila cuyo conjunto subyacente de elementos incluya a un elemento de la forma $A \rightarrow \cdot$ **error** α. Después, el analizador sintáctico "desplaza" un token ficticio **error** hacia la pila, como si hubiera visto el token **error** en su entrada.

Cuando α es ϵ, se realiza una reducción a A de inmediato y se invoca la acción semántica asociada con la producción $A \rightarrow \cdot$ **error** (que podría ser una rutina de recuperación de errores especificada por el usuario). Después, el analizador sintáctico descarta los símbolos de entrada hasta que encuentra uno con el cual pueda continuar el análisis sintáctico normal.

Si α no está vacía, `Yacc` sigue recorriendo la entrada, ignorando los símbolos hasta que encuentra una subcadena que pueda reducirse a α. Si α consiste sólo en terminales, entonces busca esta cadena de terminales en la entrada y los "reduce" al desplazarlas hacia la pila. En este punto, el analizador sintáctico tendrá a **error** α en la parte superior de su pila. Después, el analizador sintáctico reducirá **error** α a A y continuará con el análisis sintáctico normal.

Por ejemplo, una producción de error de la siguiente forma:

```
%{
#include <ctype.h>
#include <stdio.h>
#define YYSTYPE double  /* tipo double para la pila de Yacc */
%}
%token NUMERO

%left '+' '-'
%left '*' '/'
%right UMENOS
%%

lineas : lineas expr '\n'   { printf("%g\n", $2); }
       | lineas '\n'
       | /* vacia */
       | error '\n' { yyerror("reintroduzca linea anterior:");
                      yyerrok; }

       ;
expr   : expr '+' expr      { $$ = $1 + $3; }
       | expr '-' expr      { $$ = $1 - $3; }
       | expr '*' expr      { $$ = $1 * $3; }
       | expr '/' expr      { $$ = $1 / $3; }
       | '(' expr ')'       { $$ = $2; }
       | '-' expr  %prec UMENOS { $$ = - $2; }
       | NUMERO
       ;
%%
#include "lex.yy.c"
```

Figura 4.61: Calculadora de escritorio con recuperación de errores

$$instr \rightarrow \textbf{error} \ ;$$

especificaría al analizador sintáctico que debe omitir lo que esté más allá después del siguiente punto y coma al ver un error, y debe suponer que se ha encontrado una instrucción. La rutina semántica para esta producción de error no tendría que manipular la entrada, pero podría generar un mensaje de diagnóstico y establecer una bandera para inhibir la generación de código objeto, por ejemplo.

Ejemplo 4.70: La figura 4.61 muestra la calculadora de escritorio `Yacc` de la figura 4.59, con la siguiente producción de error:

```
lineas : error '\n'
```

Esta producción de error hace que la calculadora de escritorio suspenda el análisis sintáctico normal al encontrar un error sintáctico en una línea de entrada. Al encontrar el error, el ana-

lizador sintáctico en la calculadora de escritorio empieza a sacar símbolos de su pila hasta que encuentra un estado con una acción de desplazamiento en el token **error**. El estado 0 es un estado de este tipo (en este ejemplo, es el único estado así), ya que sus elementos incluyen:

$$\textit{lineas} \rightarrow \cdot \textbf{error } \texttt{'\textbackslash n'}$$

Además, el estado 0 siempre se encuentra en la parte inferior de la pila. El analizador sintáctico desplaza el token **error** hacia la pila y después continúa ignorando símbolos en la entrada hasta encontrar un carácter de nueva línea. En este punto, el analizador sintáctico desplaza el carácter de nueva línea hacia la pila, reduce **error** `'\n'` a *lineas*, y emite el mensaje de diagnóstico "reintroduzca linea anterior:". La rutina especial de `Yacc` llamada `yyerrok` restablece el analizador sintáctico a su modo normal de operación. □

4.9.5 Ejercicios para la sección 4.9

! **Ejercicio 4.9.1:** Escriba un programa en `Yacc` que reciba expresiones booleanas como entrada [según lo indicado por la gramática del ejercicio 4.2.2(g)] y produzca el valor verdadero de las expresiones.

! **Ejercicio 4.9.2:** Escriba un programa en `Yacc` que reciba listas (según lo definido por la gramática del ejercicio 4.2.2(e), pero con cualquier carácter individual como elemento, no sólo *a*) y produzca como salida una representación lineal de la misma lista; por ejemplo, una lista individual de los elementos, en el mismo orden en el que aparecen en la entrada.

! **Ejercicio 4.9.3:** Escriba un programa en `Yacc` que indique si su entrada es un *palíndromo* (secuencia de caracteres que se leen igual al derecho y al revés).

!! **Ejercicio 4.9.4:** Escriba un programa en `Yacc` que reciba expresiones regulares (según lo definido por la gramática del ejercicio 4.2.2(d), pero con cualquier carácter individual como argumento, no sólo *a*) y produzca como salida una tabla de transición para un autómata finito no determinista que reconozca el mismo lenguaje.

4.10 Resumen del capítulo 4

- *Analizadores sintácticos.* Un analizador sintáctico recibe como entrada tokens del analizador léxico, y trata los nombres de los tokens como símbolos terminales de una gramática libre de contexto. Después, el analizador construye un árbol de análisis sintáctico para su secuencia de tokens de entrada; el árbol de análisis sintáctico puede construirse en sentido figurado (pasando por los pasos de derivación correspondientes) o en forma literal.
- *Gramáticas libres de contexto.* Una gramática especifica un conjunto de símbolos terminales (entradas), otro conjunto de no terminales (símbolos que representan construcciones sintácticas) y un conjunto de producciones, cada una de las cuales proporciona una forma en la que se pueden construir las cadenas representadas por un no terminal, a partir de símbolos terminales y cadenas representados por otros no terminales. Una producción consiste en un encabezado (el no terminal a sustituir) y un cuerpo (la cadena de símbolos gramaticales de sustitución).

- *Derivaciones*. Al proceso de empezar con el no terminal inicial de una gramática y sustituirlo en forma repetida por el cuerpo de una de sus producciones se le conoce como derivación. Si siempre se sustituye el no terminal por la izquierda (o por la derecha), entonces a la derivación se le llama por la izquierda (o respectivamente, por la derecha).
- *Árboles de análisis sintáctico*. Un árbol de análisis sintáctico es una imagen de una derivación, en la cual hay un nodo para cada no terminal que aparece en la derivación. Los hijos de un nodo son los símbolos mediante los cuales se sustituye este no terminal en la derivación. Hay una correspondencia de uno a uno entre los árboles de análisis sintáctico, las derivaciones por la izquierda y las derivaciones por la derecha de la misma cadena de terminales.
- *Ambigüedad*. Una gramática para la cual cierta cadena de terminales tiene dos o más árboles de análisis sintáctico distintos, o en forma equivalente, dos o más derivaciones por la izquierda, o dos o más derivaciones por la derecha, se considera ambigua. En la mayoría de los casos de interés práctico, es posible rediseñar una gramática ambigua de tal forma que se convierta en una gramática sin ambigüedad para el mismo lenguaje. No obstante, las gramáticas ambiguas con ciertos trucos aplicados nos llevan algunas veces a la producción de analizadores sintácticos más eficientes.
- *Análisis sintáctico descendente* y *ascendente*. Por lo general, los analizadores sintácticos se diferencian en base a si trabajan de arriba hacia abajo (si empiezan con el símbolo inicial de la gramática y construyen el árbol de análisis sintáctico partiendo de la parte superior) o de abajo hacia arriba (si empiezan con los símbolos terminales que forman las hojas del árbol de análisis sintáctico y construyen el árbol partiendo de la parte inferior). Los analizadores sintácticos descendentes incluyen los analizadores sintácticos con descenso recursivo y LL, mientras que las formas más comunes de analizadores sintácticos ascendentes son analizadores sintácticos LR.
- *Diseño de gramáticas*. A menudo, las gramáticas adecuadas para el análisis sintáctico descendente son más difíciles de diseñar que las utilizadas por los analizadores sintácticos ascendentes. Es necesario eliminar la recursividad por la izquierda, una situación en la que un no terminal deriva a una cadena que empieza con el mismo no terminal. También debemos factorizar por la izquierda; las producciones de grupo para el mismo no terminal que tengan un prefijo común en el cuerpo.
- *Analizadores sintácticos de descenso recursivo*. Estos analizadores sintácticos usan un procedimiento para cada no terminal. El procedimiento analiza su entrada y decide qué producción aplicar para su no terminal. Los terminales en el cuerpo de la producción se relacionan con la entrada en el momento apropiado, mientras que las no terminales en el cuerpo producen llamadas a su procedimiento. El rastreo hacia atrás, en el caso de cuando se elige la producción incorrecta, es una posibilidad.
- *Analizadores sintácticos LL(1)*. Una gramática en la que es posible elegir la producción correcta con la cual se pueda expandir un no terminal dado, con solo analizar el siguiente símbolo de entrada, se conoce como LL(1). Estas gramáticas nos permiten construir una tabla de análisis sintáctico predictivo que proporcione, para cada no terminal y cada símbolo de preanálisis, la elección de la producción correcta. La corrección de errores se puede facilitar al colocar las rutinas de error en algunas, o en todas las entradas en la tabla que no tengan una producción legítima.

- *Análisis sintáctico de desplazamiento-reducción.* Por lo general, los analizadores sintácticos ascendentes operan mediante la elección, en base al siguiente símbolo de entrada (símbolo de anticipación) y el contenido de la pila, de si deben desplazar la siguiente entrada hacia la pila, o reducir algunos símbolos en la parte superior de la misma. Un paso de reducción toma un cuerpo de producción de la parte superior de la pila y lo sustituye por el encabezado de la producción.
- *Prefijos viables.* En el análisis sintáctico de desplazamiento-reducción, el contenido de la pila siempre es un prefijo viable; es decir, un prefijo de cierta forma de frase derecha que termina a la derecha, no más allá del final del mango de ésta. El mango es la subcadena que se introdujo en el último paso de la derivación por la derecha de esa forma de frase.
- *Elementos válidos.* Un elemento es una producción con un punto en alguna parte del cuerpo. Un elemento es válido para un prefijo viable si la producción de ese elemento se utiliza para generar el mango, y el prefijo viable incluye todos esos símbolos a la izquierda del punto, pero no los que están abajo.
- *Analizadores sintácticos LR.* Cada uno de los diversos tipos de analizadores sintácticos LR opera construyendo primero los conjuntos de elementos válidos (llamados estados LR) para todos los prefijos viables posibles, y llevando el registro del estado para cada prefijo en la pila. El conjunto de elementos válidos guía la decisión de análisis sintáctico de desplazamiento-reducción. Preferimos reducir si hay un elemento válido con el punto en el extremo derecho del cuerpo, y desplazamos el símbolo de anticipación hacia la pila si ese símbolo aparece justo a la derecha del punto, en algún elemento válido.
- *Analizadores sintácticos LR simples.* En un analizador sintáctico SLR, realizamos una reducción implicada por un elemento válido con un punto en el extremo derecho, siempre y cuando el símbolo de anticipación pueda seguir el encabezado de esa producción en alguna forma de frase. La gramática es SLR, y este método puede aplicarse si no hay conflictos de acciones de análisis sintáctico; es decir, que para ningún conjunto de elementos y para ningún símbolo de anticipación haya dos producciones mediante las cuales se pueda realizar una reducción, ni exista la opción de reducir o desplazar.
- *Analizadores sintácticos LR canónicos.* Esta forma más compleja de analizador sintáctico LR utiliza elementos que se aumentan mediante el conjunto de símbolos de anticipación que pueden seguir el uso de la producción subyacente. Las reducciones sólo se eligen cuando hay un elemento válido con el punto en el extremo derecho, y el símbolo actual de anticipación es uno de los permitidos para este elemento. Un analizador sintáctico LR canónico puede evitar algunos de los conflictos de acciones de análisis sintáctico que están presentes en los analizadores sintácticos SLR; pero a menudo tiene más estados que el analizador sintáctico SLR para la misma gramática.
- *Analizadores sintácticos LR con lectura anticipada.* Los analizadores sintácticos LALR ofrecen muchas de las ventajas de los analizadores sintácticos SLR y LR canónicos, mediante la combinación de estados que tienen los mismos corazones (conjuntos de elementos, ignorando los conjuntos asociados de símbolos de anticipación). Por ende, el número de estados es el mismo que el del analizador sintáctico SLR, pero algunos conflictos de acciones de análisis sintáctico presentes en el analizador sintáctico SLR pueden eliminarse en el analizador sintáctico LALR. Los analizadores sintácticos LALR se han convertido en el método más usado.

- *Análisis sintáctico ascendente de gramáticas ambiguas.* En muchas situaciones importantes, como en el análisis sintáctico de expresiones aritméticas, podemos usar una gramática ambigua y explotar la información adicional, como la precedencia de operadores, para resolver conflictos entre desplazar y reducir, o entre la reducción mediante dos reducciones distintas. Por ende, las técnicas de análisis sintáctico LR se extienden a muchas gramáticas ambiguas.
- `Yacc`. El generador de analizadores sintácticos `Yacc` recibe una gramática (posiblemente) ambigua junto con la información de resolución de conflictos, y construye los estados del LALR. Después produce una función que utiliza estos estados para realizar un análisis sintáctico ascendente y llama a una función asociada cada vez que se realiza una reducción.

4.11 Referencias para el capítulo 4

El formalismo de las gramáticas libres de contexto se originó con Chomsky [5], como parte de un estudio acerca del lenguaje natural. La idea también se utilizó en la descripción sintáctica de dos de los primeros lenguajes: Fortran por Backus [2] y Algol 60 por Naur [26]. El erudito Panini ideó una notación sintáctica equivalente para especificar las reglas de la gramática Sanskrit entre los años 400 a.C. y 200 a.C. [19].

Cantor [4] y Floyd [13] fueron los primeros que observaron el fenómeno de la ambigüedad. La Forma Normal de Chomsky (ejercicio 4.4.8) proviene de [6]. La teoría de las gramáticas libres de contexto se resume en [17].

El análisis sintáctico de descenso recursivo fue el método preferido para los primeros compiladores, como [16], y los sistemas para escribir compiladores, como META [28] y TMG [25]. Lewis y Stearns [24] introdujeron las gramáticas LL. El ejercicio 4.4.5, la simulación en tiempo lineal del descenso recursivo, proviene de [3].

Una de las primeras técnicas de análisis sintáctico, que se debe a Floyd [14], implicaba la precedencia de los operadores. Wirth y Weber [29] generalizaron la idea para las partes del lenguaje que no involucran operadores. Estas técnicas se utilizan raras veces hoy en día, pero podemos verlas como líderes en una cadena de mejoras para el análisis sintáctico LR.

Knuth [22] introdujo los analizadores sintácticos LR, y las tablas de análisis sintáctico LR canónicas se originaron ahí. Este método no se consideró práctico, debido a que las tablas de análisis sintáctico eran más grandes que las memorias principales de las computadoras típicas de esa época, hasta que Korenjak [23] proporcionó un método para producir tablas de análisis sintáctico de un tamaño razonable para los lenguajes de programación comunes. DeRemer desarrolló los métodos LALR [8] y SLR [9] que se usan en la actualidad. La construcción de las tablas de análisis sintáctico LR para las gramáticas ambiguas provienen de [1] y [12].

El generador `Yacc` de Johnson demostró con mucha rapidez la habilidad práctica de generar analizadores sintácticos con un generador de analizadores sintácticos LALR para los compiladores de producción. El manual para el generador de analizadores sintácticos `Yacc` se encuentra en [20]. La versión de código-abierto, `Bison`, se describe en [10]. Hay un generador de analizadores sintácticos similar llamado `CUP` [18], el cual se basa en LALR y soporta acciones escritas en Java. Los generadores de analizadores sintácticos descendentes incluyen a `Antlr` [27], un generador de analizadores sintácticos de descenso recursivo que acepta acciones en C++, Java o C#, y `LLGen` [15], que es un generador basado en LL(1).

Dain [7] proporciona una bibliografía acerca del manejo de errores sintácticos.

El algoritmo de análisis sintáctico de programación dinámica de propósito general descrito en el ejercicio 4.4.9 lo inventaron en forma independiente J. Cocke (sin publicar), Younger [30] y Kasami [21]; de aquí que se le denomine "algoritmo CYK". Hay un algoritmo más complejo de propósito general que creó Earley [11], que tabula los elementos LR para cada subcadena de la entrada dada; este algoritmo, que también requiere un tiempo $O(n^3)$ en general, sólo requiere un tiempo $O(n^2)$ en las gramáticas sin ambigüedad.

1. Aho, A. V., S. C. Johnson y J. D. Ullman, "Deterministic parsing of ambiguous grammars", *Comm. ACM* **18**:8 (Agosto, 1975), pp. 441-452.

2. Backus, J. W, "The syntax and semantics of the proposed international algebraic language of the Zurich-ACM-GAMM Conference", *Proc. Intl. Conf. Information Processing*, UNESCO, París (1959), pp. 125-132.

3. Birman, A. y J. D. Ullman, "Parsing algorithms with backtrack", *Information and Control* **23**:1 (1973), pp. 1-34.

4. Cantor, D. C., "On the ambiguity problem of Backus systems", *J. ACM* **9**:4 (1962), pp. 477-479.

5. Chomsky, N., "Three models for the description of language", *IRE Trans. on Information Theory* **IT-2**:3 (1956), pp. 113-124.

6. Chomsky, N., "On certain formal properties of grammars", *Information and Control* **2**:2 (1959), pp. 137-167.

7. Dain, J., "Bibliography on Syntax Error Handling in Language Translation Systems", 1991. Disponible en el grupo de noticias `comp.compilers`; vea `http://compilers.iecc.com/comparch/article/91-04-050`.

8. DeRemer, F., "Practical Translators for LR(k) Languages", Tésis Ph.D., MIT, Cambridge, MA, 1969.

9. DeRemer, F., "Simple LR(k) grammars", *Comm. ACM* **14**:7 (Julio, 1971), pp. 453-460.

10. Donnelly, C. y R. Stallman, "Bison: The YACC-compatible Parser Generator", `http://www.gnu.org/software/bison/manual/`.

11. Earley, J., "An efficient context-free parsing algorithm", *Comm. ACM* **13**:2 (Febrero, 1970), pp. 94-102.

12. Earley, J., "Ambiguity and precedence in syntax description", *Acta Informatica* **4**:2 (1975), pp. 183-192.

13. Floyd, R. W., "On ambiguity in phrase-structure languages", *Comm. ACM* **5**:10 (Octubre, 1962), pp. 526-534.

14. Floyd, R. W., "Syntactic analysis and operator precedence", *J. ACM* **10**:3 (1963), pp. 316-333.

15. Grune, D. y C. J. H. Jacobs, "A programmer-friendly LL(1) parser generator", *Software Practice and Experience* **18**:1 (Enero, 1988), pp. 29-38. Vea también `http://www.cs.vu.nl/~ceriel/LLgen.html`.

16. Hoare, C. A. R., "Report on the Elliott Algol translator", *Computer J.* **5**:2 (1962), pp. 127-129.

17. Hopcroft, J. E., R. Motwani y J. D. Ullman, *Introduction to Automata Theory, Languages, and Computation*, Addison-Wesley, Boston, MA, 2001.

18. Hudson, S. E. *et al.*, "CUP LALR Parser Generator in Java", Disponible en `http://www2.cs.tum.edu/projects/cup/`.

19. Ingerman, P. Z., "Panini-Backus form suggested", *Comm. ACM* **10**:3 (Marzo, 1967), p. 137.

20. Johnson, S. C., "Yacc — Yet Another Compiler Compiler", Computing Science Technical Report 32, Bell Laboratories, Murray Hill, NJ, 1975. Disponible en `http://dinosaur.compilertools.net/yacc/`.

21. Kasami, T., "An efficient recognition and syntax analysis algorithm for context-free languages", AFCRL-65-758, Air Force Cambridge Research Laboratory, Bedford, MA, 1965.

22. Knuth, D. E., "On the translation of languages from left to right", *Information and Control* **8**:6 (1965), pp. 607-639.

23. Korenjak, A. J., "A practical method for constructing LR(k) processors", *Comm. ACM* **12**:11 (Noviembre, 1969), pp. 613-623.

24. Lewis, P. M. II y R. E. Stearns, "Syntax-directed transduction", *J. ACM* **15**:3 (1968), pp. 465-488.

25. McClure, R. M., "TMG — a syntax-directed compiler", *proc. 20th ACM Natl. Conf.* (1965), pp. 262-274.

26. Naur, P. *et al.*, "Report on the algorithmic language ALGOL 60", *Comm. ACM* **3**:5 (Mayo, 1960), pp. 299-314. Vea también *Comm. ACM* **6**:1 (Enero, 1963), pp. 1-17.

27. Parr, T., "ANTLR", `http://www.antlr.org/`.

28. Schorre, D. V., "Meta-II: a syntax-oriented compiler writing language", *Proc. 19th ACM Natl. Conf.* (1964), pp. D1.3-1–D1.3.-11.

29. Wirth, N. y H. Weber, "Euler: a generalization of Algol and its formal definition: Part I", *Comm. ACM* **9**:1 (Enero, 1966), pp. 13-23.

30. Younger, D. H., "Recognition and parsing of context-free languages in time n^3", *Information and Control* **10**:2 (1967), pp. 189-208.