UNAH
UNIVERSIDAD NACIONAL
AUTÓNOMA DE HONDURAS

Trabajo de Investigación

Análisis sintáctico

Facultad de Ingeniería

Departamento de Ingeniería en Sistemas

Choluteca, Honduras

Miércoles 31 de Julio 2024

UNAH-CURLP
CENTRO UNIVERSITARIO REGIONAL
DEL LITORAL PACÍFICO

UNIVERSIDAD NACIONAL AUTÓNOMA DE HONDURAS

CENTRO UNIVERSITARIO REGIONAL DEL LITORAL PACÍFICO

Facultad de Ingeniería

Departamento de Ingeniería en Sistemas

IS-913 Diseño de Compiladores

Ing. Oscar Omar Pineda

Equipo #2

Allan Josue Campos

David Isaac Rivera

Elvis Jose Murillo

Nataly Jasmin Alvarez

Omri David Aguilar

Ramon Edgardo Pastrana

Choluteca, Honduras Miércoles 31 de Julio 2024

Tabla de contenido

Tabla de ilustraciones

Introducción

El análisis sintáctico es una etapa crucial en la compilación de lenguajes de programación, que se encarga de validar y estructurar el código fuente según las reglas sintácticas del lenguaje.

El presente informe tiene como objetivo proporcionar un análisis exhaustivo de los diferentes aspectos del análisis sintáctico en el ámbito de la teoría de lenguajes y compiladores. Se aborda inicialmente la función del analizador sintáctico, discutiendo su importancia y los métodos para manejar errores sintácticos, así como las estrategias para la recuperación de dichos errores. Se exploran las gramáticas libres de contexto, detallando su definición formal, las derivaciones, los árboles de análisis sintáctico, la ambigüedad y la verificación del lenguaje generado por una gramática. Además, se realiza una comparación entre las gramáticas libres de contexto y las expresiones regulares, de igual manera se incluye una sección dedicada a la escritura de una gramática, donde se compara el análisis léxico con el análisis sintáctico, se discute la eliminación de la ambigüedad y se presentan construcciones de lenguajes que no son libres de contexto.

En cuanto al análisis sintáctico descendente, se describen métodos como el análisis de descenso recursivo, los conjuntos PRIMERO y SIGUIENTE, las gramáticas LL(1) y el análisis predictivo no recursivo, incluyendo la recuperación de errores en este contexto. Por otro lado, el análisis sintáctico ascendente se aborda a través de la explicación de conceptos como las reducciones, la poda de mangos, el análisis de desplazamiento-reducción y los conflictos que pueden surgir durante este proceso.

Se introduce el análisis sintáctico LR, comenzando con los analizadores SLR (LR simple), y se explica la importancia de los analizadores LR, los elementos y el autómata LR(0), el algoritmo de análisis sintáctico LR, la construcción de tablas de análisis SLR y los prefijos viables, y se profundiza en los analizadores sintácticos LR más avanzados, incluyendo los elementos LR(1) canónicos, la construcción de conjuntos de elementos LR(1), las tablas de análisis LR(1) canónico y las tablas LALR, así como la compactación eficiente de estas tablas.

Finalmente, se discute el uso de gramáticas ambiguas, resolviendo conflictos mediante precedencia y asociatividad, y se aborda la ambigüedad del “else colgante” y la recuperación de errores en el análisis sintáctico LR. La última sección está dedicada a los generadores de analizadores sintácticos, en particular el generador Yacc, su uso con gramáticas ambiguas y la recuperación de errores en Yacc.

Objetivos

General

- Analizar y comprender los diferentes tipos de analizadores sintácticos, su funcionamiento y su aplicación en el procesamiento de lenguajes, destacando su importancia y sus características dentro de la teoría de gramáticas libres de contexto.

Específicos

1. Examinar y clasificar los distintos tipos de analizadores sintácticos, tanto descendentes como ascendentes, describiendo sus métodos, ventajas, y limitaciones en el contexto del procesamiento de lenguajes formales.
2. Estudiar y explicar la estructura y los componentes de las gramáticas libres de contexto, incluyendo símbolos terminales, no terminales, y producciones, así como su papel fundamental en la derivación y construcción de árboles de análisis sintáctico.
3. Identificar y discutir los problemas de ambigüedad en las gramáticas y las técnicas utilizadas para resolverlos, tales como la eliminación de la recursividad por la izquierda y la factorización por la izquierda, además de explorar la eficiencia de los analizadores sintácticos LR y sus variantes (SLR, LALR, y LR canónicos).

Resumen Ejecutivo

Un analizador sintáctico es un componente clave en los sistemas de procesamiento de lenguajes formales, responsable de recibir una secuencia de tokens del analizador léxico y construir un árbol de análisis sintáctico basado en una gramática libre de contexto. Las gramáticas libres de contexto definen las reglas del lenguaje mediante un conjunto de símbolos terminales, no terminales, y producciones, permitiendo la derivación y representación de secuencias válidas de símbolos.

El proceso de derivación reemplaza iterativamente un no terminal inicial con producciones adecuadas, lo cual puede ser visualizado a través de un árbol de análisis sintáctico. La ambigüedad en las gramáticas puede causar múltiples árboles de análisis para una misma secuencia de entrada, un problema que puede ser mitigado mediante el rediseño de la gramática.

Existen dos enfoques principales para el análisis sintáctico: descendente y ascendente. Los analizadores sintácticos descendentes, como los analizadores recursivos y LL, construyen el árbol de análisis de arriba hacia abajo. Los analizadores sintácticos ascendentes, como los analizadores LR, construyen el árbol de abajo hacia arriba, guiados por conjuntos de elementos válidos que informan las decisiones de desplazamiento y reducción. Las variantes de analizadores LR (SLR, LALR y LR canónicos) ofrecen diferentes balances entre complejidad y eficiencia.

Herramientas como Yacc facilitan la creación de analizadores sintácticos LALR a partir de gramáticas potencialmente ambiguas, utilizando información adicional como la precedencia de operadores para resolver conflictos de análisis. En conclusión, los analizadores sintácticos son fundamentales en la interpretación y procesamiento de lenguajes, y una comprensión detallada de su funcionamiento y diseño es esencial para cualquier estudio en el campo de la teoría de lenguajes formales y la compilación.

Análisis sintáctico

1.1 Introducción

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.

1.1.1 La función del analizador sintáctico

Todo lenguaje de programación obedece a unas reglas que describen la estructura sintáctica de los programas bien formados que acepta. En Pascal, por ejemplo, un programa se compone de bloques; un bloque, de sentencias; una sentencia, de expresiones; una expresión, de componentes léxicos; y así sucesivamente hasta llegar a los caracteres básicos.

La sintaxis de las construcciones de los lenguajes de programación se puede describir mediante gramáticas de contexto libre o utilizando la notación BNF (Backus-Naur Form). Estas herramientas son fundamentales para definir las reglas sintácticas que un compilador debe seguir para analizar y procesar correctamente el código fuente.

Dramáticas de contexto libre

- Son particularmente útiles porque permiten describir de manera precisa y formal la estructura de los lenguajes de programación.

La notación BNF

- Es una forma de representar estas gramáticas de manera compacta y legible.

En nuestro modelo de compilador, el analizador sintáctico juega un papel crucial al recibir una cadena de tokens proporcionada por el analizador léxico. El objetivo principal del analizador sintáctico es verificar que la secuencia de tokens recibida pueda ser generada mediante la gramática del lenguaje fuente. Esta verificación es esencial para asegurar que el programa está correctamente estructurado según las reglas del lenguaje de programación.

Una de las expectativas clave del analizador sintáctico es que sea capaz de reportar cualquier error sintáctico de manera clara y comprensible. Además, es importante que el

analizador pueda recuperarse de errores comunes para poder continuar procesando el resto del programa. Esta capacidad de recuperación permite que el compilador pueda manejar programas con errores sin detenerse abruptamente.

Para los programas bien formados, el analizador sintáctico construye un árbol de análisis sintáctico (también conocido como árbol de derivación). Este árbol representa la estructura jerárquica del programa y es fundamental para las etapas posteriores del compilador. Sin embargo, en la práctica, no siempre es necesario construir este árbol de manera explícita. Las acciones de comprobación y traducción pueden intercalarse con el análisis sintáctico, lo que optimiza el proceso y reduce la necesidad de estructuras de datos adicionales.

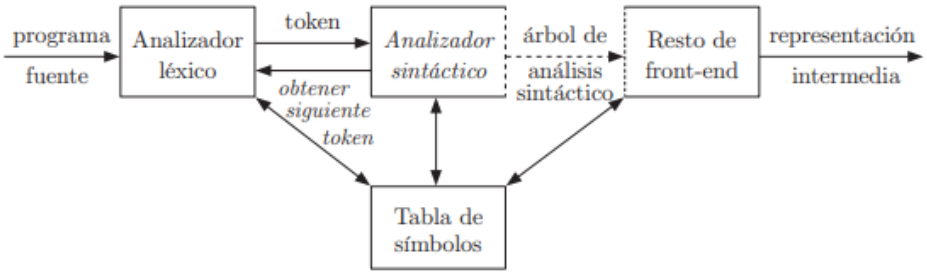


Ilustración 1 Posición del analizador sintáctico en el modelo del compilador

Existen tres tipos generales de analizadores para las gramáticas:

Métodos universales

- El algoritmo de Cocke-Younger Kasami y el algoritmo de Earley pueden analizar cualquier gramática

Métodos descendentes

- Construyen árboles de análisis sintáctico desde la parte superior (raíz) hacia la parte inferior (hojas)

Métodos ascendentes

- Empiezan desde las hojas y avanzan hacia la raíz. En ambos casos, la entrada al analizador se explora de izquierda a derecha, un símbolo a la vez.

Los métodos descendentes y ascendentes más eficientes solo funcionan para subclases específicas de gramáticas. Entre estas, las gramáticas LL y LR son lo suficientemente expresivas para describir la mayoría de las construcciones sintácticas en los lenguajes de programación modernos.

Las gramáticas LL (Left-to-right, Leftmost derivation) son aquellas donde el análisis se realiza de izquierda a derecha y se produce la derivación más a la izquierda. Este tipo de gramática es adecuada para los métodos de análisis sintáctico predictivo, que suelen implementarse manualmente en los compiladores.

Por otro lado, las gramáticas LR (Left-to-right, Rightmost derivation) son aquellas donde el análisis también se realiza de izquierda a derecha, pero se produce la derivación más a la derecha de una forma no determinista. Los analizadores LR son más potentes y pueden manejar una clase más extensa de gramáticas. Generalmente, estos analizadores se construyen mediante herramientas automatizadas debido a la complejidad de su implementación manual.

1.1.2 Manejo de los errores sintácticos

Si un compilador tuviera que procesar solo programas correctos, su diseño e implementación se simplificarían considerablemente. 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 estos 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, y su manejo eficaz es crucial para proporcionar una buena experiencia de desarrollo. Los compiladores deben estar equipados con estrategias robustas para la detección y recuperación de errores sintácticos. Las estrategias de recuperación en modo de pánico y a nivel de frase son métodos populares utilizados en la práctica.

Léxicos

- Producidos al escribir mal un identificador, una palabra clave o un operador.

Sintácticos

- Por una expresión aritmética o paréntesis no equilibrados.

Semánticos

- Como un operador aplicado a un operando incompatible

Lógicos

- Puede ser una llamada infinitamente recursiva.

De corrección

- Cuando el programa no hace lo que el programador realmente deseaba.

1.1.3 Estrategias para recuperarse de los errores

Una vez que se detecta un error, ¿cómo debe recuperarse el analizador sintáctico? Aunque no existe una estrategia universalmente aceptada, 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. Sin embargo, esto puede no ser lo más eficiente para el programador.

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. Para lograr esto, es importante implementar estrategias de recuperación que permitan al analizador continuar trabajando después de detectar un error. estrategias de recuperación de los errores:

Recuperación en modo de pánico

- Con este método, al detectar un error, el analizador sintáctico descarta los símbolos de entrada uno a uno hasta encontrar un conjunto designado de tokens de sincronización. Generalmente, los tokens de sincronización son delimitadores como el punto y coma o }, cuya función en el programa fuente es clara y sin ambigüedades. Es responsabilidad del diseñador del compilador seleccionar los tokens de sincronización apropiados para el lenguaje fuente. Aunque la corrección en modo de pánico puede omitir una cantidad considerable de entrada sin verificar errores adicionales, tiene la ventaja de ser simple y 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, sustituyendo un prefijo de la entrada 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, quien debe tener cuidado de no crear ciclos infinitos, como sucedería si siempre se insertara algo delante del símbolo de entrada actual.
- La sustitución a nivel de frase se ha utilizado en varios compiladores que reparan errores, ya que puede corregir cualquier cadena de entrada. Su desventaja principal es la dificultad para manejar 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.

1.2 Gramáticas libres de contexto

Las gramáticas libres de contexto (CFG, por sus siglas en inglés) son fundamentales en el análisis sintáctico de los lenguajes de programación. Estas gramáticas definen reglas para la estructura de los programas y permiten que los compiladores validen y procesen el código fuente. A continuación, se desglosan los conceptos esenciales relacionados con las CFGs.

1.2.1 La definición formal de una gramática libre de contexto

Una gramática libre de contexto (CFG) se define formalmente como un conjunto de cuatro componentes:

- Los terminales (T): son los símbolos básicos a partir de los cuales se forman las cadenas. El término "nombre de token" es un sinónimo 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. los terminales son las palabras reservadas ***if*** y ***else***, y los símbolos "**(**" y "**)**".
- Los no terminales (N): son variables sintácticas que denotan conjuntos de cadenas. 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.
- Un símbolo inicial (S): el conjunto de cadenas que denota es el lenguaje generado por la gramática. Por convención, las producciones para el símbolo inicial se listan primero, En una gramática, un no terminal se distingue como el símbolo inicial.
- 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:
    - Un no terminal, conocido como encabezado o lado izquierdo de la producción; esta producción define algunas de las cadenas denotadas por el encabezado.
    - El símbolo →. Algunas veces se ha utilizado :: = en vez de la flecha.

- 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.

1.2.2 Derivaciones

La construcción de un árbol de análisis sintáctico puede realizarse de manera precisa si consideramos una perspectiva derivacional, donde las producciones se tratan como reglas de reescritura. Comenzando con el símbolo inicial, cada paso de reescritura sustituye un no terminal por el cuerpo de una de sus producciones.

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

$$E \rightarrow E + E \mid E * E \mid - E \mid (E) \mid id$$

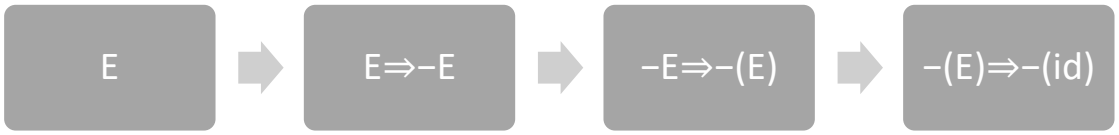
La producción $E \rightarrow - E$ significa que, si E denota una expresión, entonces $- E$ debe también denotar una expresión. La sustitución de una sola E por $- E$ se describirá escribiendo lo siguiente:

$$E \Rightarrow -E$$

o 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,

$$E \Rightarrow -E \Rightarrow -(E) \Rightarrow -(id)$$

A dicha secuencia de sustituciones la llamamos una derivación de −(id) a partir de E. Esta derivación proporciona la prueba de que la cadena −(id) es una instancia específica de una expresión.



Este proceso de aplicar producciones para transformar E en −(id) ilustra cómo las gramáticas libres de contexto pueden describir y generar las estructuras de un lenguaje de programación.

1.2.3 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.

En el primer paso de la derivación, E ⇒ −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 ⇒ −(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.

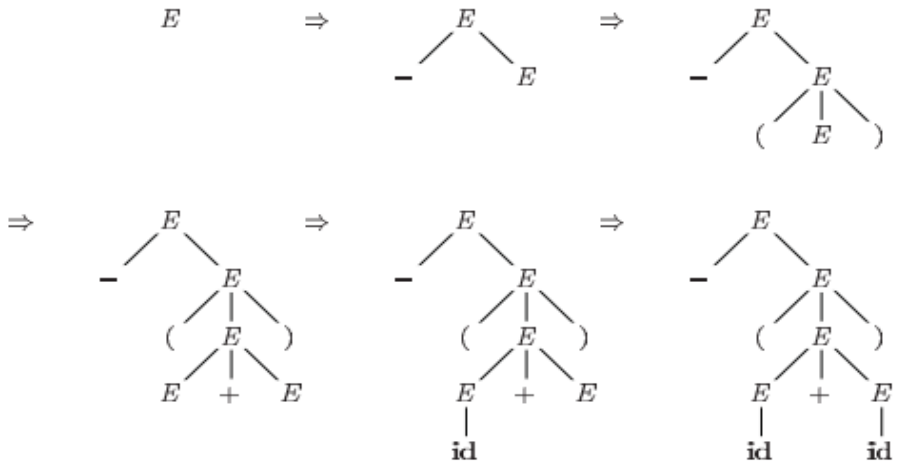


Ilustración 2 Secuencia de árboles de análisis sintáctico

1.2.4 Ambigüedad

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.

La gramática de expresiones aritméticas permite dos derivaciones por la izquierda distintas para el enunciado id + id ∗ id:

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

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.

1.2.5 Verificación del lenguaje generado por una gramática

Verificar el lenguaje generado por una gramática implica determinar si una gramática dada genera todas y solo las cadenas del lenguaje que describe. Esto se hace mediante:

- Derivaciones: Comprobando que todas las cadenas del lenguaje pueden ser derivadas usando las reglas de producción.
- Proporcionando contraejemplos: Mostrando que no se puede derivar una cadena que no pertenece al lenguaje.

La verificación es crucial para asegurarse de que la gramática describe con precisión el lenguaje de programación objetivo.

1.2.6 Comparación entre gramáticas libres de contexto y expresiones regulares

Las gramáticas libres de contexto y las expresiones regulares son herramientas fundamentales en la teoría de lenguajes formales y la construcción de compiladores, pero tienen capacidades y aplicaciones diferentes:

Expresiones regulares: Adecuadas para describir patrones de texto lineales y secuenciales, como los tokens en un lenguaje de programación. Son más simples y eficientes para el análisis léxico, pero no pueden describir estructuras jerárquicas como las de un lenguaje de programación.

Gramáticas libres de contexto: Adecuadas para describir la estructura jerárquica y anidada de los lenguajes de programación. Son más potentes que las expresiones regulares y pueden representar relaciones recursivas, pero son más complejas de analizar y procesar.

Por ejemplo, las expresiones regulares pueden describir identificadores y números, mientras que las gramáticas libres de contexto pueden describir la estructura de sentencias condicionales, bucles y funciones.

1.3 Escritura de una gramática

La escritura de una gramática es un proceso esencial en la creación de un compilador, ya que define la estructura sintáctica del lenguaje de programación. Una gramática

formalmente describe cómo se pueden formar frases válidas en el lenguaje mediante reglas de producción que transforman símbolos no terminales en secuencias de símbolos terminales y no terminales. Una gramática bien definida facilita el análisis sintáctico y garantiza que el código fuente siga las reglas sintácticas del lenguaje.

1.3.1 Comparación entre análisis léxico y análisis sintáctico

El análisis léxico y el análisis sintáctico son dos fases críticas en el proceso de compilación:

Análisis Léxico

- Esta fase se encarga de transformar la secuencia de caracteres de la entrada en una secuencia de tokens. Un token es una unidad léxica que representa elementos como palabras clave, identificadores, operadores y delimitadores. El análisis léxico utiliza expresiones regulares para identificar y clasificar estos tokens.

Análisis Sintáctico

- Esta fase toma la secuencia de tokens producida por el analizador léxico y verifica si esta secuencia puede ser generada por la gramática del lenguaje. El análisis sintáctico construye un árbol de análisis sintáctico (AST) que representa la estructura jerárquica del código fuente. Utiliza técnicas como análisis descendente y ascendente para reconocer las estructuras sintácticas.

La principal diferencia entre ambas fases es que el análisis léxico se enfoca en la estructura de los tokens individuales, mientras que el análisis sintáctico se enfoca en la estructura de alto nivel del programa.

- ❖ 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.
- ❖ Las reglas léxicas de un lenguaje son con frecuencia bastante simples, y para describirlas no necesitamos una notación tan poderosa como las gramáticas.
- ❖ Por lo general, las expresiones regulares proporcionan una notación más concisa y fácil de entender para los tokens, en comparación con las gramáticas.
- ❖ Pueden construirse analizadores léxicos más eficientes en forma automática a partir de expresiones regulares, en comparación con las gramáticas arbitrarias.

1.3.2 Eliminación de la ambigüedad

La eliminación de la ambigüedad es crucial para asegurar que el compilador interprete el código de manera consistente. Algunos métodos para eliminar la ambigüedad incluyen:

Reescribir la Gramática: Modificar las reglas de producción para que cada cadena de entrada tenga una única derivación.

Uso de Prioridades y Asociatividades: Definir reglas de precedencia y asociatividad para operadores, lo que ayuda a resolver ambigüedades en expresiones aritméticas.

Introducción de Producciones Auxiliares: Crear nuevas reglas que desambiguarán las producciones conflictivas.

1.3.3 Construcciones de lenguajes que no son libres de contexto

Los lenguajes libres de contexto son aquellos que pueden ser descritos por una gramática libre de contexto, donde cada regla de producción tiene una forma de A → α, con A como un no terminal y α como una secuencia de terminales y no terminales. Sin embargo, hay ciertas construcciones de lenguajes que no pueden ser descritas por gramáticas libres de contexto, como:

Dependencias de Contexto

- Lenguajes donde ciertas construcciones dependen del contexto en el que aparecen. Por ejemplo, lenguajes con un número igual de a y b (a^n b^n) no son libres de contexto.

Correspondencias Anidadas

- Lenguajes que requieren correspondencias anidadas, como paréntesis balanceados, que pueden ser descritos por gramáticas libres de contexto, pero otras anidaciones complejas pueden no serlo.

Dependencias de Múltiples Niveles

- Lenguajes que requieren dependencias entre múltiples niveles de anidación o estructuras que van más allá de las capacidades de las gramáticas libres de contexto.

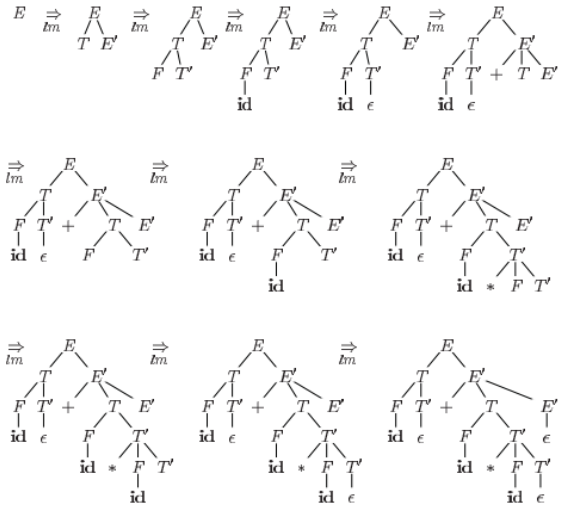


Ilustración 3 Análisis sintáctico descendente para id+id∗id

1.4 Análisis sintáctico descendente

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

Por ejemplo, considere el análisis sintáctico descendente en la figura, 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′ → +TE′; en el segundo nodo E′, se elige la producción E′ → ε. 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).

1.4.1 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 */;
        }
    }
```

Ilustración 4 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 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 este 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 o el método de Earley.

Para permitir el rastreo hacia atrás, hay que modificar el código de la figura. 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.

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.

1.4.2 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.

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.

Definimos a SIGUIENTE(A), para él 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 $S \Rightarrow^{*} \alpha A a \beta$, para algunas α y β.

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.

- Si X es un terminal, entonces PRIMERO(X) = {X }.
- Si X es un no terminal y X → Y1Y2 …Yk es una producción para cierta k ≥ 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(Y1), …, PRIMERO(Y*i*−1); es decir, Y1 …Y*i*−1 ⇒∗ ε. Si ε está en PRIMERO (Y*j*) para todas las j = 1, 2, …, *k*, entonces se agrega ε a PRIMERO(X). Por ejemplo, todo lo que hay en PRIMERO(Y1) se encuentra sin duda en PRIMERO (X). Si Y1 no deriva a ε, entonces no agregamos nada más a PRIMERO (X), pero si Y1 ⇒∗ ε, entonces agregamos PRIMERO(Y2), y así sucesivamente.
- Si X → ε es una producción, entonces se agrega ε a PRIMERO (X).

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.

- Colocar $ en SIGUIENTE (S), en donde S es el símbolo inicial y $ es el delimitador derecho de la entrada.
- Si hay una producción A → αBβ, entonces todo lo que hay en PRIMERO(β) excepto ε está en SIGUIENTE(B).
- Si hay una producción A → αB, o una producción A → αBβ, en donde PRIMERO(β) contiene a ε, entonces todo lo que hay en SIGUIENTE(A) está en SIGUIENTE(B).

1.4.3 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.

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).

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:

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

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.

1.4.4 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 controlado por una tabla, que se muestra en la figura, tiene un búfer de entrada, una pila que contiene una secuencia de símbolos gramaticales, una tabla de análisis sintáctico construida, 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.

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.

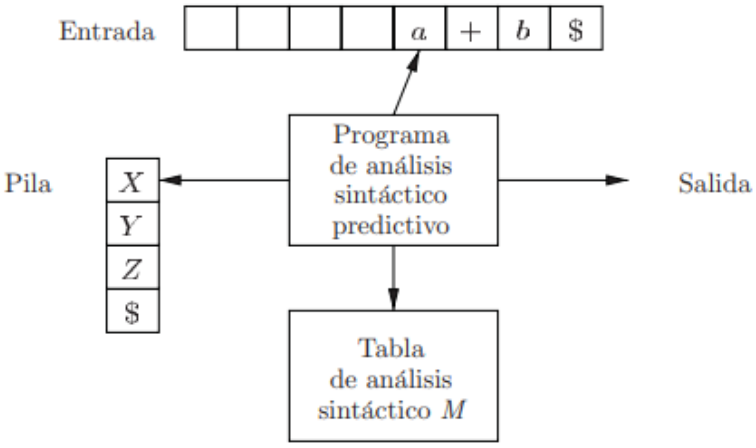


Ilustración 5 Modelo de un analizador sintáctico predictivo, controlado por una tabla

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 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 $Y_k, Y_{k-1}, \ldots, Y_1$ en la pila, con Y_1 en la parte superior;
    }
    establecer *X* con el símbolo de la parte superior de la pila;
}

Ilustración 6 Algoritmo de análisis sintáctico predictivo

1.4.5 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:

Como punto inicial, colocar todos los símbolos que están en SIGUIENTE(A) en el conjunto de sincronización para el no terminal A.

No basta con usar SIGUIENTE(A) como el conjunto de sincronización para A.

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).

Si un no terminal puede generar la cadena vacía, entonces la producción que deriva a ε puede usarse como predeterminada.

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.

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.

1.5 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.

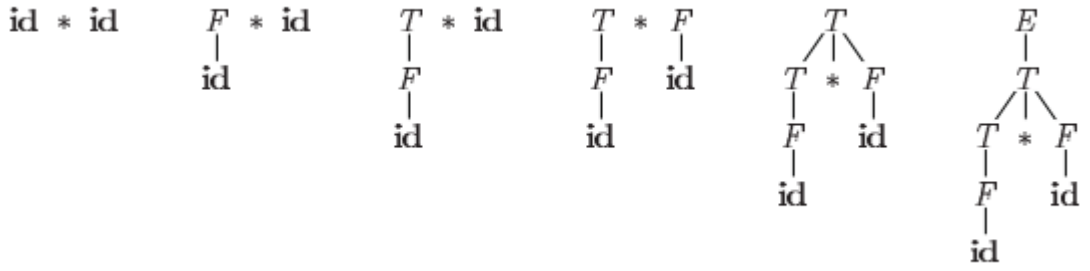


*Ilustración 7 Un análisis sintáctico ascendente para id * id*

1.5.1 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.

1.5.2 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 id1 ∗ id2, de acuerdo con la gramática de expresiones, son como en la figura 4.26. Aunque T es el cuerpo de la producción E → T, el símbolo T no es un mango en la forma de frase T ∗ id2. Si T se sustituyera por E, obtendríamos la cadena E ∗ id2, 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
$\mathbf{id}_1 * \mathbf{id}_2$	$\mathbf{id}_1$	$F \to \mathbf{id}$
$F * \mathbf{id}_2$	F	$T \to F$
$T * \mathbf{id}_2$	$\mathbf{id}_2$	$F \to \mathbf{id}$
$T * F$	$T * F$	$E \to T * F$

*Ilustración 8 Mangos durante un análisis sintáctico de id1 * id2*

1.5.3 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 id1 * id2.

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

*Ilustración 9 Configuraciones de un analizador sintáctico de desplazamiento-reducción, con una entrada id1 * id2*

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.

Desplazar

- Desplazar el siguiente símbolo de entrada y lo coloca en la parte superior de la pila.

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.

Aceptar

- Anunciar que el análisis sintáctico se completó con éxito.

Error

- Descubrir un error de sintaxis y llamar a una rutina de recuperación de errores.

1.5.4 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).

1.6 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.

1.6.1 ¿Por qué analizadores sintácticos 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:

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 aun así puede implementarse con la misma eficiencia que otros métodos más primitivos de desplazamiento-reducción.
	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.

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.

1.6.2 Los elementos y el autómata LR(0)

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.

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). En especial, cada estado del autómata LR(0) representa un conjunto de elementos en la colección LR(0) canónica.

- 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.

- 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).

- 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.

- 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.

1.6.3 El algoritmo de análisis sintáctico LR

En la figura 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.

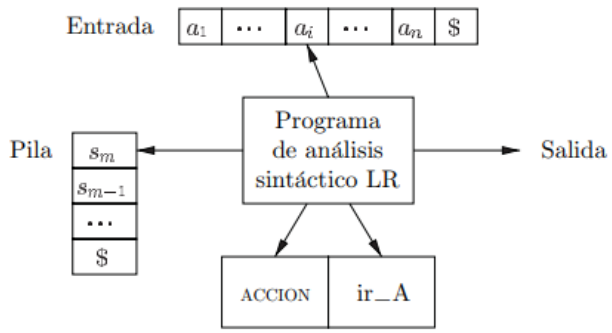


Ilustración 10 Modelo de un analizador sintáctico LR

- 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 → β. 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.

2. Extendemos la función ir_A, definida en los conjuntos de elementos, a los estados: si ir_A[Ii, A] = Ij, entonces ir_A también asigna un estado i y un no terminal A al estado j.

1.6.4 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). 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		$\mathbf{id} * \mathbf{id} + \mathbf{id} \$$	desplazar
(2)	0 5	$\mathbf{id}$	$* \mathbf{id} + \mathbf{id} \$$	reducir mediante $F \to \mathbf{id}$
(3)	0 3	F	$* \mathbf{id} + \mathbf{id} \$$	reducir mediante $T \to F$
(4)	0 2	T	$* \mathbf{id} + \mathbf{id} \$$	desplazar
(5)	0 2 7	$T *$	$\mathbf{id} + \mathbf{id} \$$	desplazar
(6)	0 2 7 5	$T * \mathbf{id}$	$+ \mathbf{id} \$$	reducir mediante $F \to \mathbf{id}$
(7)	0 2 7 10	$T * F$	$+ \mathbf{id} \$$	reducir mediante $T \to T * F$
(8)	0 2	T	$+ \mathbf{id} \$$	reducir mediante $E \to T$
(9)	0 1	E	$+ \mathbf{id} \$$	desplazar
(10)	0 1 6	$E +$	$\mathbf{id} \$$	desplazar
(11)	0 1 6 5	$E + \mathbf{id}$	$\$$	reducir mediante $F \to \mathbf{id}$
(12)	0 1 6 3	$E + F$	$\$$	reducir mediante $T \to F$
(13)	0 1 6 9	$E + T$	$\$$	reducir mediante $E \to E + T$
(14)	0 1	E	$\$$	aceptar

*Ilustración 11 Movimientos de un analizador sintáctico LR con id * id + id*

1.6.5 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.

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.

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, 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.

1.7 Analizadores sintácticos LR más poderosos

Entre los métodos que utilizan esos analizadores tenemos:

Método "LR canónico", o simplemente "LR"	Utiliza al máximo el (los) símbolo(s) de preanálisis. Este método utiliza un extenso conjunto de elementos, conocidos como elementos LR(1).
Método "LR con símbolo de preanálisis" o "LALR(lookahead LR)"	Se basa en los conjuntos de elementos LR(0), y tiene mucho menos estados que los analizadores sin tácticos 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.

1.7.1 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 *Ii* 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: Consideremos la siguiente gramática:

$$S \rightarrow B\ B$$

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

Hay una derivación por la derecha $S * \Rightarrow aaBab \Rightarrow aaaBab$. 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 $S * \Rightarrow BaB \Rightarrow BaaB$. De esta derivación podemos ver que el elemento $[B \rightarrow \alpha\cdot B, \$]$ es válido para el prefijo viable Baa.

1.7.2 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;

}

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 → α·Bβ, a] en el conjunto de elementos válido para cierto prefijo viable γ. Entonces hay una derivación por la derecha S ∗ ⇒ δAax ⇒ δαBβax, en donde γ = δα. Suponga que βax deriva a la cadena de terminales by. Entonces, para cada producción de la forma B → η para cierta η, tenemos la derivación S ∗ ⇒ γBby ⇒ γηby. Por ende, [B →·η, b] es válida para γ. Observe que b puede ser el primer ter- minal derivado a partir de β, o que es posible que β derive a en la derivación βax ∗ ⇒ 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).

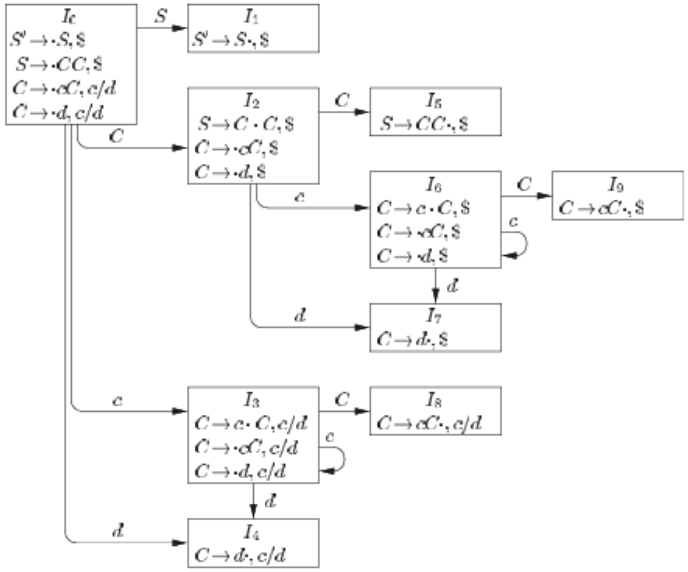


Ilustración 12 El gráfico de ir_A

1.7.3 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: 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: Para construir un analizador sintáctico LR canónico, primero se genera la colección de conjuntos de elementos LR(1) para la gramática. Luego, se definen las acciones de análisis para cada estado: "desplazar" si un elemento tiene una producción con un símbolo terminal siguiente, "reducir" si la producción está completa y "aceptar" para el símbolo de fin de cadena. Si surgen conflictos de acción, la gramática no es LR(1). Se construyen las transiciones para los no terminales, y las entradas no definidas se configuran como "error". Finalmente, se establece el estado inicial del analizador a partir del conjunto que contiene la producción inicial con el símbolo de fin de cadena.

La tabla que se forma a partir de la acción de análisis sintáctico y las funciones producidas por el Algoritmo 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.

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		

Ilustración 13 Tabla de análisis sintáctico canónica

1.7.4 Construcción de tablas de análisis sintáctico LALR

La técnica LALR (LR con lectura anticipada) se utiliza frecuentemente en la construcción de analizadores sintácticos debido a que genera tablas significativamente más pequeñas que las tablas LR canónicas, permitiendo manejar la mayoría de las construcciones sintácticas comunes de los lenguajes de programación. Aunque las tablas SLR y LALR suelen tener el mismo número de estados, las tablas LALR son preferidas porque pueden manejar construcciones que las técnicas SLR no pueden.

Al combinar estados con corazones comunes (primeros componentes de los elementos LR), se puede reducir el número de estados sin introducir conflictos de desplazamiento/reducción. Este proceso implica unir conjuntos de elementos con el mismo corazón, lo que permite construir un analizador más compacto y eficiente sin perder la precisión en la mayoría de los casos.

Por ejemplo, al unir los estados I4 e I7 en I47, el nuevo estado reduce en cualquier entrada, manteniendo el comportamiento del analizador original y atrapando errores antes de que se desplacen más símbolos de entrada. Este método se extiende a otros estados con corazones comunes, permitiendo combinar transacciones y funciones activas sin crear nuevos conflictos, siempre que la gramática original no los tenga.

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

ENTRADA: Una gramática aumentada G.

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

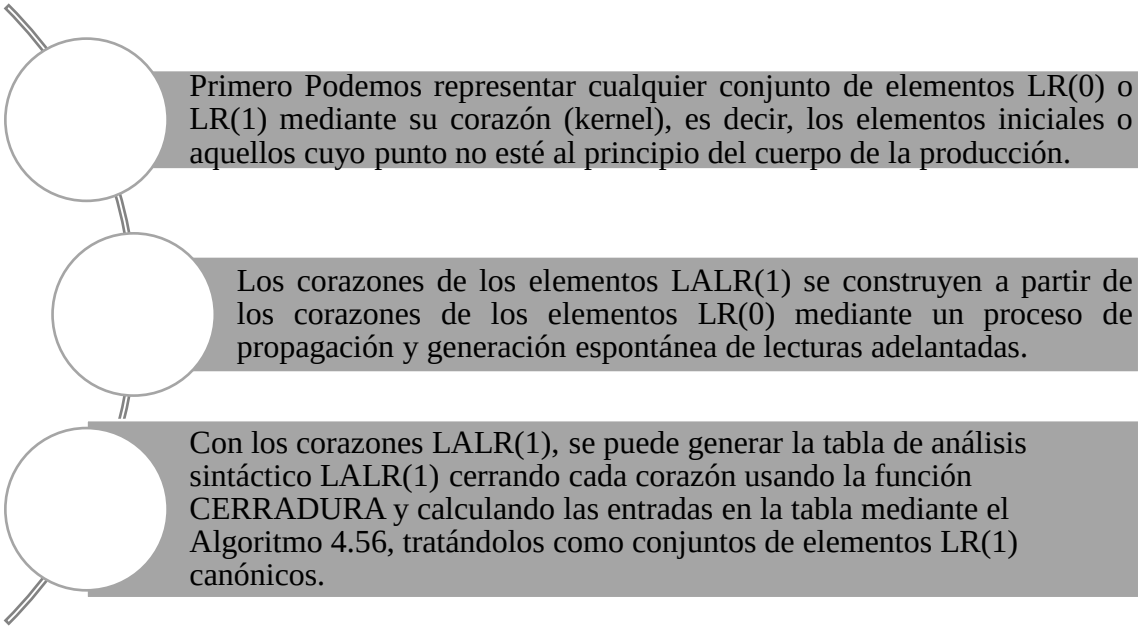
MÉTODO: Para construir un analizador sintáctico LALR, primero se genera la colección de conjuntos de elementos LR(1). Luego, se identifican los conjuntos con el mismo corazón y se sustituyen por su unión. Los conjuntos resultantes, {J0, J1, …, Jm}, se utilizan para determinar las acciones de análisis sintáctico de la misma manera que en el método LR canónico. Si hay un conflicto de acciones, la gramática no es LALR(1) y el algoritmo no produce un analizador. La tabla de transiciones se construye combinando las transiciones de los conjuntos unidos, asegurando que los corazones de los conjuntos combinados sean iguales.

A la tabla producida por este algoritmo 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).

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		

Ilustración 14 Tabla de análisis sintáctico LALR

1.7.5 Construcción eficiente de tablas de análisis sintáctico LALR



1.7.6 Compactación de las tablas de análisis sintáctico LR

Una gramática de lenguaje de programación típica con 50 a 100 terminales y 100 producciones puede generar una tabla de análisis sintáctico LALR con cientos de estados y una función de acción con alrededor de 20,000 entradas, cada una requiriendo al menos 8 bits para codificarse. En dispositivos pequeños, es crucial usar una codificación más eficiente que un arreglo bidimensional. Una técnica efectiva para compactar el campo de acción es identificar y reutilizar filas idénticas de la tabla, asignando un apuntador unidimensional a estados con acciones idénticas, y utilizando el terminal como un desplazamiento a partir del apuntador del estado. Otra técnica, que mejora la eficiencia del espacio a expensas de una ligera reducción en la velocidad, es crear listas de acciones para cada estado, con pares de (terminal, acción) y una acción predeterminada para entradas no encontradas. Las entradas de error pueden sustituirse por acciones de reducción para mantener la uniformidad y detectar errores antes de un desplazamiento.

1.8 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 una forma estrictamente controlada; de no ser así, no puede haber garantía en el lenguaje que reconozca un analizador sintáctico.

1.8.1 Precedencia y asociatividad para resolver conflictos

La gramática ambigua para expresiones con los operadores + y $*$ no especifica la asociatividad ni la precedencia de estos operadores. La gramática sin ambigüedad resuelve esto otorgando menor precedencia a + que a $*$ y haciendo que ambos sean asociativos por la izquierda. Sin embargo, la gramática ambigua es preferida porque permite cambiar la asociatividad y precedencia sin alterar las producciones o el número de estados en el analizador sintáctico.

La gramática ambigua genera conflictos en las acciones de análisis sintáctico en los estados que corresponden a los conjuntos de elementos I7 e I8. Estos conflictos pueden resolverse utilizando la información sobre precedencia y asociatividad de + y $*$. Por ejemplo,

si ∗ tiene precedencia sobre +, el analizador debería desplazar ∗ hacia la pila, mientras que si + tiene precedencia sobre ∗, debería reducir E + E a E. La asociatividad también influye en la resolución de conflictos; si + es asociativo por la izquierda, se debería reducir mediante E → E + E cuando se encuentra un +.

Al asumir que + es asociativo por la izquierda y ∗ tiene precedencia sobre +, se puede construir una tabla de análisis sintáctico LR que maneje adecuadamente los conflictos de la gramática ambigua. Esta tabla se puede comparar con la obtenida eliminando las reducciones mediante producciones simples en una gramática sin ambigüedad. Las gramáticas ambiguas se pueden manejar de manera similar en los contextos de 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	

Ilustración 15 Tabla de análisis sintáctico

1.8.2 La ambigüedad del "else colgante"

La gramática para las instrucciones condicionales presenta una ambigüedad conocida como el "else colgante". La gramática original:

instr → if expr then instr else instr

| if expr then instr

| otras

es ambigua porque no especifica claramente con cuál if se asocia cada else.

Para simplificar, se usa una abstracción:

$$S \rightarrow i\ S\ e\ S \mid i\ S \mid a$$

donde i representa if expr then, e representa else y a representa otras producciones. La ambigüedad se manifiesta en un conflicto de desplazamiento/reducción en el estado I4, donde:

$$S \rightarrow i\ S \cdot e\ S$$

$$S \rightarrow i\ S \cdot$$

hay un conflicto sobre si desplazar e o reducir S → i S.

La resolución correcta es desplazar else (e) para asociarlo con el then (i) más cercano. Esto asegura que el else se asocie correctamente con el if correspondiente. La tabla de análisis sintáctico SLR resultante se muestra en la figura 4.51 y resuelve el conflicto a favor del desplazamiento en la entrada e.

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	

Ilustración 16 Tabla de análisis sintáctico LR para la gramática del "else colgante"

1.8.3 Recuperación de errores en el análisis sintáctico LR

La recuperación de errores en el análisis sintáctico LR se realiza detectando errores cuando se encuentra una entrada de error en la tabla de acciones de análisis sintáctico. Los errores se detectan tan pronto como no hay una continuación válida para la porción de la entrada que se ha explorado hasta ese momento. Un analizador sintáctico LR canónico detecta errores sin realizar reducciones, mientras que los analizadores SLR y LALR pueden hacer varias reducciones antes de anunciar un error.

En el modo de pánico para la recuperación de errores en el análisis sintáctico LR, se explora la pila hasta encontrar un estado con una transición válida en un no terminal específico. Luego, se descartan símbolos de entrada hasta encontrar uno que pueda seguir legítimamente al no terminal. Este método intenta eliminar la frase que contiene el error.

Para una recuperación a nivel de frase, se examinan las entradas de error en la tabla de análisis y se deciden las acciones correctivas basadas en el uso del lenguaje. Las acciones pueden incluir la inserción o eliminación de símbolos de la pila o de la entrada, o la alteración de los símbolos de entrada. Estas rutinas de manejo de errores se diseñan para evitar ciclos infinitos y asegurar que se realicen avances en el análisis.

Un ejemplo con la gramática de expresiones muestra cómo se modifican las entradas de error en la tabla de análisis sintáctico para realizar reducciones específicas o llamar a rutinas de error que diagnostican y corrigen errores comunes, como operadores faltantes o paréntesis desbalanceados.

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	

Ilustración 17 Tabla de análisis sintáctico LR con rutinas de error

1.9 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.

1.9.1 El generador de analizadores sintácticos Yacc

Yacc (Yet Another Compiler-Compiler) es una herramienta que se utiliza para construir traductores. A continuación, se describe el proceso:

Preparación del Archivo de Especificación

Se crea un archivo con la especificación de Yacc, por ejemplo, traducir.y.

Generación del Analizador Sintáctico:

Se ejecuta el comando de UNIX: yacc traducir.y.

Esto transforma traducir.y en un programa en C llamado y.tab.c utilizando el método LALR.

Compilación del Programa:

Se compila y.tab.c junto con la biblioteca ly que contiene el programa de análisis sintáctico LR mediante el comando: cc y.tab.c -ly.

Esto produce el programa objeto a.out, que realiza la traducción especificada por el programa original en Yacc.

El archivo fuente de Yacc tiene tres partes principales:

Declaraciones	Reglas de Traducción	Soporte de Rutinas en C
• Secciones iniciales donde se declaran tokens y otros elementos.	• Definiciones de las reglas gramaticales.	• Código en C que apoya la funcionalidad del analizador.

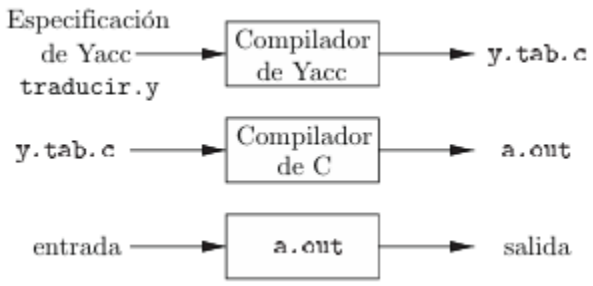


Ilustración 18 Creación de un traductor de entrada/salida con Yacc

Ejemplo: Calculadora de Escritorio con Yacc

Se describe la creación de una calculadora de escritorio simple usando Yacc. La calculadora evalúa expresiones aritméticas y las imprime. La gramática utilizada incluye producciones para sumas, multiplicaciones, paréntesis y dígitos.

Estructura del programa Yacc:

1. **Parte de las Declaraciones:**
   - Declaraciones en C y de tokens.
   - Ejemplo: #include <ctype.h> y %token DIGITO.
2. **Parte de las Reglas de Traducción:**
   - Definición de reglas y acciones semánticas asociadas.
   - Ejemplo: expr : expr '+' term { $$ = $1 + $3; } | term ;
3. **Parte de Soporte en C:**
   - Incluye el analizador léxico yylex().
   - yylex() lee caracteres y devuelve tokens como DIGITO.

Proceso Completo:

1. **Archivo de Especificación (traducir.y)**:
   - Se define la gramática y las acciones semánticas.
2. **Generación del Analizador (yacc traducir.y)**:
   - Produce y.tab.c que contiene el analizador.
3. **Compilación del Analizador (cc y.tab.c -ly)**:
   - Produce el ejecutable a.out.

Ejemplo de Especificación Yacc:

```
%{
#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;
}
```

1.9.2 Uso de Yacc con gramáticas ambiguas

Se modifica la especificación de Yacc para crear una calculadora de escritorio más funcional que pueda evaluar una secuencia de expresiones, permitiendo líneas en blanco entre ellas. La nueva regla es:

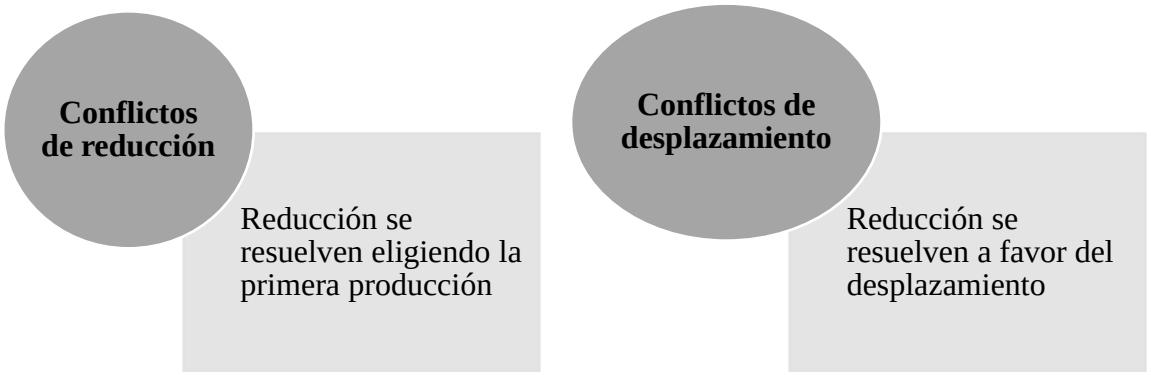
```
lineas : lineas expr '\n' { printf("%g\n", $2); }
       | lineas '\n'
       | /* vacía */;
```

Esta regla permite que la calculadora procese múltiples expresiones en secuencia y tolere líneas en blanco.

Expansión de la Gramática: Se amplía la clase de expresiones para incluir números completos y operadores aritméticos (+, -, *, /) así como el operador unario -.

```
E : E + E | E - E | E * E | E / E | - E | numero;
```

Manejo de Conflictos de Análisis Sintáctico: Debido a la ambigüedad de la gramática, Yacc puede generar conflictos de acciones de análisis sintáctico. Yacc utiliza dos reglas predeterminadas para resolver estos conflictos:



Asignación de Precedencias y Asociatividades: Se pueden asignar precedencias y asociatividades a los terminales para resolver conflictos de manera específica:

- *%left '+' '-' hace que + y - sean asociativos a la izquierda.*
- *%right '^' hace que ^ sea asociativo a la derecha.*
- *%nonassoc '<' define operadores no asociativos.*

Ejemplo de Producción con Precedencia Forzada: Para manejar precedencias en producciones específicas, se utiliza %prec:

```
expr : '-' expr %prec UMENOS;
```

Aquí, UMENOS se usa para definir la precedencia del operador unario -.

Especificación Completa Modificada:

```
%{
#include <ctype.h>
%}
%token DIGITO
%left '+' '-'
%left '*' '/'
%right UMENOS
```

```
%%
lineas : lineas expr '\n' { printf("%g\n", $2); }
       | lineas '\n'
       | /* vacía */
       ;
expr   : expr '+' expr { $$ = $1 + $3; }
       | expr '-' expr { $$ = $1 - $3; }
       | expr '*' expr { $$ = $1 * $3; }
       | expr '/' expr { $$ = $1 / $3; }
       | '-' expr %prec UMENOS { $$ = -$2; }
       | DIGITO
       ;
%%
yylex() {
    int c;
    c = getchar();
    if (isdigit(c)) {
        yylval = c - '0';
        return DIGITO;
    }
    return c;
}
```

Esta especificación permite que la calculadora procese una secuencia de expresiones, soporte líneas en blanco, y maneje operadores aritméticos y unarios de manera adecuada mediante la asignación de precedencias y asociatividades.

1.9.4 Recuperación de errores en Yacc

Yacc maneja la recuperación de errores utilizando producciones de error. El proceso general es el siguiente:

Selección de No Terminales Importantes

- Elegir no terminales que generen expresiones, instrucciones, bloques y funciones.

Producciones de Error

- Agregar producciones de error en la forma A → error α.

Manejo de Errores

- Yacc desplaza un token error y procesa de manera especial según la producción de error.

Ejemplo de Producción de Error

- lineas : error '\n' { yyerror("reintroduzca linea anterior:"); yyerrok; }
- Maneja errores en líneas de entrada y emite mensajes de diagnóstico.

Resolución de Conflictos

- La producción de error permite suspender el análisis normal y recuperar el estado del análisis con yyerrok

Conclusiones

El análisis sintáctico es una etapa esencial en la compilación de programas. Comprender su funcionamiento es vital para diseñar compiladores y herramientas de procesamiento de lenguajes.

El manejo de errores sintácticos y las estrategias de recuperación son fundamentales para crear compiladores robustos que puedan proporcionar retroalimentación útil al programador, mejorando la calidad del software.

Conocer las gramáticas libres de contexto y la escritura de gramáticas permite optimizar el análisis y la traducción de código, haciéndolo más eficiente y preciso.

La capacidad de escribir y manipular gramáticas es crucial para el diseño de nuevos lenguajes de programación y lenguajes específicos de dominio, permitiendo personalizar herramientas según las necesidades específicas.

Diferenciar entre análisis sintáctico descendente y ascendente, y conocer técnicas como el análisis de descenso recursivo, LL(1), LR, y LALR, permite seleccionar la estrategia más adecuada para el problema en cuestión, optimizando el rendimiento del analizador.

El manejo de gramáticas ambiguas y la resolución de conflictos de precedencia y asociatividad son habilidades cruciales para garantizar que los analizadores sintácticos interpreten correctamente el código fuente.

El uso de generadores de analizadores sintácticos, como Yacc, facilita la creación automática de analizadores, ahorrando tiempo y esfuerzo en el desarrollo de compiladores.

Una comprensión profunda de los elementos teóricos, como las derivaciones, árboles de análisis, y el algoritmo LR, junto con la aplicación práctica de estos conceptos, es esencial para cualquier profesional en el campo de la ingeniería de software y la computación.

Referencias

Aho, A. V., Lam, M. S., Sethi, R., & Ullman, J. D. (2008). *Compiladores: Principios, técnicas y herramientas* (Segunda ed.). (L. M. Castillo, Ed., & A. V. Elizondo, Trad.) Naucalpan de Juárez, Mexico: PEARSON EDUCACIÓN. Recuperado el Julio de 2024