Análisis sintáctico

Pedro O. Pérez M., PhD

Desarrollo de aplicaciones avanzadas de ciencias computacionales
Módulo 3: Compiladores
Tecnológico de Monterrey

pperezm@tec.mx

05-2023

1. 4.1 - 4.3
   - 4.1 Introducción
   - 4.2 Gramáticas libres de contexto
   - 4.3 Escribiendo una gramática

2. 4.4 - 4.6
   - 4.4 Analisis Top-Down
   - 4.5 Análisis Bottom-Up
   - 4.6 Analizadores LR

3. 4.7 - 4.9
   - 4.7 Analizadores LR más poderosos
   - 4.9 Generadores de analizadores sintácticos

La sintaxis de las construcciones de un lenguaje de programa deben ser especificados a través de una gramática libre de contexto o en la notación BNF (Backus-Naur Form). Pero, ¿cuáles son sus beneficios?

- Una gramática da un especificación precisa y fácil de entender de un lenguaje de programación.
- Para ciertas gramáticas, nosotros podemos construir automáticamente un analizar sintáctico eficiente; con el beneficio adicional de que el proceso de construcción nos puede revelar ambigüedades sintácticas.
- La estructura de una gramática es útil para una correcta traducción de código fuente a código objeto y para la detección de errores.
- La gramática permite que un lenguaje se desarrolle o evolucione iterativamente.

Existen tres tipos generales de analizadores sintácticos:

- Universal (Algoritmos Cocke-Younger-Kasami y Earley).
- Top-down (LL).
- Bottom-up (LR).

En cualquier caso, la entrada del analizador sintáctico es revisada de izquierda a derecha, un símbolo a la vez.

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

Ejemplo de gramática LR

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

Ejemplo de gramática LL

Los errores de programación comúnmente pueden ocurrir en los siguientes niveles:

- Errores léxicos.
- Errores sintácticos.
- Errores semánticos.
- Errores lógicos.

El manejador de errores de un analizador sintáctico tiene metas sencillas de describir pero difíciles de realizar:

- Reportar la presencia de errores de forma clara y acertada.
- Recuperarse de los errores rápidamente y así poder detectar los siguientes errores.
- Agregar la mínima sobrecarga al procesamiento de programas correctos.

Estrategias de recuperación de errores

Existen 4 estrategias principales para la recuperación de errores:

- Modo pánico: Cuando se descubre un error, el analizador sintáctico descarta todos los tokens de entrada hasta encontrar un *token de sincronización*.
- Nivel de frase: Reemplazar un prefijo de la entrada restante por alguna cadena que permita al analizador continuar.
- Producciones erróneas.
- Corrección global.

Definición formar de una gramática libre de contexto

Un gramática libre de contexto consiste de:

- **Terminales.** Son los símbolos básicos de cuales están formados las cadenas (strings).
- **No terminales**. Son las variables sintácticas que hacen referencia a un conjunto de cadenas.
- **Símbolo inicial**.
- **Producciones**. Una producción consiste en;
  - Un no terminal llamado *cabeza* o *lado izquierdo*.
  - El símbolo $\rightarrow$.
  - Un *cuerpo* o *lado derecho* que consiste en cero o más terminales y no terminales.

$$
\begin{aligned}
\textit{expression} &\rightarrow \textit{expression} + \textit{term} \\
\textit{expression} &\rightarrow \textit{expression} - \textit{term} \\
\textit{expression} &\rightarrow \textit{term} \\
\textit{term} &\rightarrow \textit{term} * \textit{factor} \\
\textit{term} &\rightarrow \textit{term} / \textit{factor} \\
\textit{term} &\rightarrow \textit{factor} \\
\textit{factor} &\rightarrow (\ \textit{expression}\ ) \\
\textit{factor} &\rightarrow \mathbf{id}
\end{aligned}
$$

Figure 4.2: Grammar for simple arithmetic expressions

1. These symbols are terminals:
   (a) Lowercase letters early in the alphabet, such as a, b, c.
   (b) Operator symbols such as $+$, $*$, and so on.
   (c) Punctuation symbols such as parentheses, comma, and so on.
   (d) The digits $0, 1, \ldots, 9$.
   (e) Boldface strings such as **id** or **if**, each of which represents a single terminal symbol.

2. These symbols are nonterminals:

   (a) Uppercase letters early in the alphabet, such as A, B, C.

   (b) The letter S, which, when it appears, is usually the start symbol.

   (c) Lowercase, italic names such as *expr* or *stmt*.

   (d) When discussing programming constructs, uppercase letters may be used to represent nonterminals for the constructs. For example, nonterminals for expressions, terms, and factors are often represented by E, T, and F, respectively.

3. Uppercase letters late in the alphabet, such as X, Y, Z, represent *grammar symbols*; that is, either nonterminals or terminals.

4. Lowercase letters late in the alphabet, chiefly $u, v, \ldots, z$, represent (possibly empty) strings of terminals.

5. Lowercase Greek letters, α, β, γ for example, represent (possibly empty) strings of grammar symbols. Thus, a generic production can be written as $A \to \alpha$, where A is the head and α the body.

6. A set of productions $A \to \alpha_1$, $A \to \alpha_2, \ldots, A \to \alpha_k$ with a common head A (call them *A-productions*), may be written $A \to \alpha_1 \mid \alpha_2 \mid \cdots \mid \alpha_k$. Call $\alpha_1, \alpha_2, \ldots, \alpha_k$ the *alternatives* for A.

7. Unless stated otherwise, the head of the first production is the start symbol.

To understand how parsers work, we shall consider derivations in which the nonterminal to be replaced at each step is chosen as follows:

1. In *leftmost* derivations, the leftmost nonterminal in each sentential is always chosen. If $\alpha \Rightarrow \beta$ is a step in which the leftmost nonterminal in α is replaced, we write $\alpha \underset{lm}{\Rightarrow} \beta$.

2. In *rightmost* derivations, the rightmost nonterminal is always chosen; we write $\alpha \underset{rm}{\Rightarrow} \beta$ in this case.

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

$$-(\mathbf{id} + \mathbf{id})$$

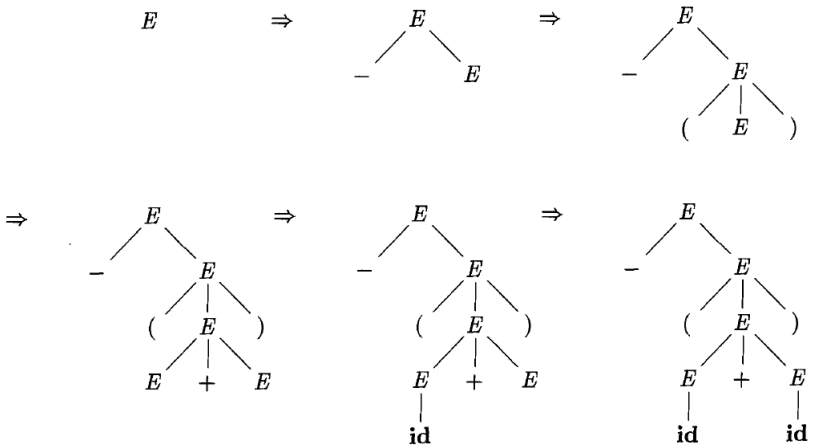

Figure 4.4: Sequence of parse trees for derivation (4.8)

Si una gramática produce más de un árbol de análisis para una misma entrada, entonces es una gramática ambigua.

Gramáticas libres de contexto vs. expresiones regulares.

Cada construcción que pueda ser descrita por una expresión regular puede ser descrita con una gramática libre de contexto, pero no lo contrario.
Alternativamente, cada lenguaje regular es un lenguaje libre de contexto, pero no lo contrario.

- Para cada estado i de un AFN, crea un no terminal A_i.
- 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 va al estado j con una transición ϵ, agrega la producción $A_i \rightarrow A_j$.
- Si i es un estado de aceptación, agrega $A_i \rightarrow \epsilon$.
- Si i es el estado inicial, haz A_i el símbolo inicial de la gramática.

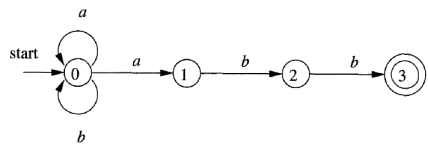

Figure 3.24: A nondeterministic finite automaton

Exercise 4.2.1 : Consider the context-free grammar:

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

and the string $aa + a*$.

a) Give a leftmost derivation for the string.

b) Give a rightmost derivation for the string.

c) Give a parse tree for the string.

! d) Is the grammar ambiguous or unambiguous? Justify your answer.

! e) Describe the language generated by this grammar.

Análisis léxico vs sintáctico

Si todo lo que puede ser descrito por una expresión regular también puede ser descrito por una gramática, ¿porqué usamos expresiones regulares para definir el análisis léxico de un lenguaje?

- Separando la estructura sintáctica de un lenguaje en un componente léxico y otro no léxico provee de una conveniente manera de modularizar el "Front End" en dos componentes manejables.
- La reglas léxicas de un lenguaje son frecuentemente muy simples, no se requiere una notación tan poderosa como las gramáticas.
- Las expresiones regulares generalmente proveen de una notación más concisa y simple para los tokens.
- Analizadores léxico eficientes pueden ser construidos automáticamente a partir de expresiones regulares que de gramáticas.

$$
\begin{array}{lcl}
stmt & \rightarrow & \textbf{if } expr \textbf{ then } stmt \\
 & | & \textbf{if } expr \textbf{ then } stmt \textbf{ else } stmt \\
 & | & \textbf{other}
\end{array}
$$

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

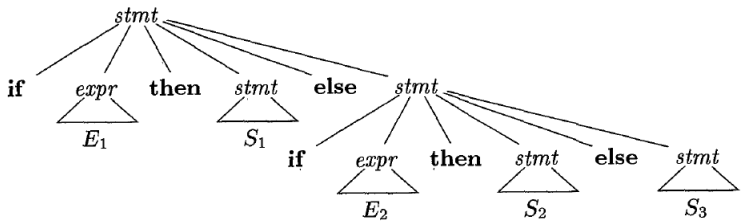

Figure 4.8: Parse tree for a conditional statement

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

has the two parse trees shown in Fig. 4.9.

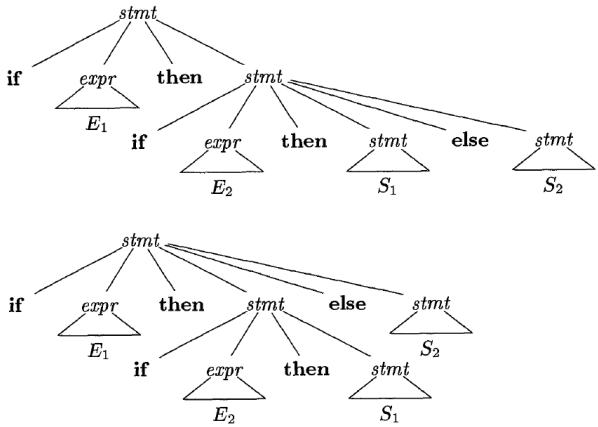

Figure 4.9: Two parse trees for an ambiguous sentence

$$
\begin{array}{rcl}
\textit{stmt} & \rightarrow & \textit{matched\_stmt} \\
 & | & \textit{open\_stmt} \\
\textit{matched\_stmt} & \rightarrow & \textbf{if } \textit{expr} \textbf{ then } \textit{matched\_stmt} \textbf{ else } \textit{matched\_stmt} \\
 & | & \textbf{other} \\
\textit{open\_stmt} & \rightarrow & \textbf{if } \textit{expr} \textbf{ then } \textit{stmt} \\
 & | & \textbf{if } \textit{expr} \textbf{ then } \textit{matched\_stmt} \textbf{ else } \textit{open\_stmt}
\end{array}
$$

Figure 4.10: Unambiguous grammar for if-then-else statements

Eliminando la recursión por la izquierda

$$A \to \beta A'$$
$$A' \to \alpha A' \mid \epsilon$$

$$\begin{aligned} E &\to E + T \mid E - T \mid T \\ T &\to T * F \mid T / F \mid F \\ F &\to (E) \mid \mathbf{id} \end{aligned}$$

Algorithm 4.19: Eliminating left recursion.

INPUT: Grammar G with no cycles or ϵ-productions.

OUTPUT: An equivalent grammar with no left recursion.

METHOD: Apply the algorithm in Fig. 4.11 to G. Note that the resulting non-left-recursive grammar may have ϵ-productions. $\square$

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

```
1)  arrange the nonterminals in some order A_1, A_2, ..., A_n.
2)  for ( each i from 1 to n ) {
3)      for ( each j from 1 to i - 1 ) {
4)          replace each production of the form A_i → A_j γ by the
                productions A_i → δ_1 γ | δ_2 γ | ··· | δ_k γ, where
                A_j → δ_1 | δ_2 | ··· | δ_k are all current A_j-productions
5)      }
6)      eliminate the immediate left recursion among the A_i-productions
7)  }
```

Figure 4.11: Algorithm to eliminate left recursion from a grammar

$$A \to \alpha\beta_1 \mid \alpha\beta_2 \mid \cdots \mid \alpha\beta_n \mid \gamma$$

$$\downarrow$$

$$\begin{aligned} A &\to \alpha A' \mid \gamma \\ A' &\to \beta_1 \mid \beta_2 \mid \cdots \mid \beta_n \end{aligned}$$

$$S \rightarrow i\ E\ t\ S \mid i\ E\ t\ S\ e\ S \mid a$$
$$E \rightarrow b$$

Exercise 4.3.1: The following is a grammar for regular expressions over symbols a and b only, using $+$ in place of $|$ for union, to avoid conflict with the use of vertical bar as a metasymbol in grammars:

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

a) Left factor this grammar.

b) Does left factoring make the grammar suitable for top-down parsing?

c) In addition to left factoring, eliminate left recursion from the original grammar.

d) Is the resulting grammar suitable for top-down parsing?

Exercise 4.3.1: The following is a grammar for regular expressions over symbols a and b only, using $+$ in place of $|$ for union, to avoid conflict with the use of vertical bar as a metasymbol in grammars:

$$\text{a) } S \rightarrow 0\,S\,1 \mid 0\,1 \text{ with string } 000111.$$

a) Left factor this grammar.

b) Does left factoring make the grammar suitable for top-down parsing?

c) In addition to left factoring, eliminate left recursion from the original grammar.

d) Is the resulting grammar suitable for top-down parsing?

Exercise 4.3.1: The following is a grammar for regular expressions over symbols a and b only, using $+$ in place of $|$ for union, to avoid conflict with the use of vertical bar as a metasymbol in grammars:

$$\textbf{!}\ \text{c)}\ \ S \rightarrow S\,(\,S\,)\,S \mid \epsilon \text{ with string } (()()).$$

a) Left factor this grammar.

b) Does left factoring make the grammar suitable for top-down parsing?

c) In addition to left factoring, eliminate left recursion from the original grammar.

d) Is the resulting grammar suitable for top-down parsing?

Revisar la sección 4.4.1 del libro de [AHO]

- FIRST(α), donde α es cualquier cadena de símbolos de la gramática, es el conjunto de terminales que empieza una secuencia de cadenas que derivan de α.
- FOLLOW(A), para un no terminal A, es el conjunto de terminales a que pueden aparecer inmediatamente a la derecha de A en alguna sentencia.

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

1. If X is a terminal, then $\text{FIRST}(X) = \{X\}$.
2. If X is a nonterminal and $X \rightarrow Y_1 Y_2 \cdots Y_k$ is a production for some $k \geq 1$, then place a in $\text{FIRST}(X)$ if for some i, a is in $\text{FIRST}(Y_i)$, and ϵ is in all of $\text{FIRST}(Y_1), \ldots, \text{FIRST}(Y_{i-1})$; that is, $Y_1 \cdots Y_{i-1} \stackrel{*}{\Rightarrow} \epsilon$. If ϵ is in $\text{FIRST}(Y_j)$ for all $j = 1, 2, \ldots, k$, then add ϵ to $\text{FIRST}(X)$. For example, everything in $\text{FIRST}(Y_1)$ is surely in $\text{FIRST}(X)$. If Y_1 does not derive ϵ, then we add nothing more to $\text{FIRST}(X)$, but if $Y_1 \stackrel{*}{\Rightarrow} \epsilon$, then we add $\text{FIRST}(Y_2)$, and so on.
3. If $X \rightarrow \epsilon$ is a production, then add ϵ to $\text{FIRST}(X)$.

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

1. Place \$ in FOLLOW(S), where S is the start symbol, and \$ is the input right endmarker.
2. If there is a production $A \rightarrow \alpha B \beta$, then everything in FIRST(β) except ϵ is in FOLLOW(B).
3. If there is a production $A \rightarrow \alpha B$, or a production $A \rightarrow \alpha B \beta$, where FIRST($\beta$) contains ϵ, then everything in FOLLOW(A) is in FOLLOW(B).

- Los analizadores predictivos, o analizadores de descenso recursivo sin "backtracking" pueden ser construidos para una clase de gramáticas conocidas como LL(1).
- Una gramática G es LL(1), sí y solo si siendo $A \rightarrow \alpha|\beta$ son distintas producciones de G, se dan las siguientes condiciones:
  - Para ningún terminal *a*, ambas cadenas empiezan con el mismo símbolo.
  - A lo mas un no terminal, α o β, pueden derivar en ϵ.
  - Si $\beta \Rightarrow \epsilon$, entonces α no debe derivar ninguna cadena que empiece con una terminal en FOLLOW(A). De igual forma, $\alpha \Rightarrow \epsilon$, entonces β no puede derivar ninguna cadena que empiece con un termina en FOLLOW(A).

Ejercicio 4.3

Para la siguiente gramática, genera:

- Los conjuntos de FIRST y FOLLOW.
- ¿Es una gramática LL(1)?

$$
\begin{aligned}
\mathrm{S} &\rightarrow \mathrm{A}\ \mathrm{a} \\
\mathrm{A} &\rightarrow \mathrm{B}\ \mathrm{D} \\
\mathrm{B} &\rightarrow \mathrm{b} \mid \epsilon \\
\mathrm{D} &\rightarrow \mathrm{d} \mid \epsilon
\end{aligned}
$$

Algorithm 4.31 : Construction of a predictive parsing table.

INPUT: Grammar G.

OUTPUT: Parsing table M.

METHOD: For each production $A \to \alpha$ of the grammar, do the following:

1. For each terminal a in FIRST(A), add $A \to \alpha$ to $M[A, a]$.
2. If ϵ is in FIRST(α), then for each terminal b in FOLLOW(A), add $A \to \alpha$ to $M[A, b]$. If ϵ is in FIRST(α) and \$ is in FOLLOW(A), add $A \to \alpha$ to $M[A, \$]$ as well.

Ejercicio 4.4

Para la siguiente gramática:

- Genera los conjuntos de FIRST y FOLLOW.

$$
\begin{aligned}
S &\rightarrow iEtSS' \mid a \\
S' &\rightarrow eS \mid \epsilon \\
E &\rightarrow b
\end{aligned}
$$

NON - TERMINAL	INPUT SYMBOL					
	a	b	e	i	t	\$
S	$S \to a$			$S \to iEtSS'$		
S'			$S' \to \epsilon$ $S' \to eS$			$S' \to \epsilon$
E		$E \to b$				

Figure 4.18: Parsing table M for Example 4.33

METHOD: Initially, the parser is in a configuration with $w\$$ in the input buffer and the start symbol S of G on top of the stack, above \$. The program in Fig. 4.20 uses the predictive parsing table M to produce a predictive parse for the input. □

```
set ip to point to the first symbol of w;
set X to the top stack symbol;
while ( X ≠ $ ) { /* stack is not empty */
        if ( X is a ) pop the stack and advance ip;
        else if ( X is a terminal ) error();
        else if ( M[X, a] is an error entry ) error();
        else if ( M[X, a] = X → Y1 Y2 ··· Yk ) {
                output the production X → Y1 Y2 ··· Yk;
                pop the stack;
                push Yk, Yk−1, ... , Y1 onto the stack, with Y1 on top;
        }
        set X to the top stack symbol;
}
```

Figure 4.20: Predictive parsing algorithm

NON - TERMINAL	INPUT SYMBOL					
	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	$F \to \mathbf{id}$			$F \to (E)$		

Figure 4.17: Parsing table M for Example 4.32

id + id * id

Ejercicio 4.5

Para la siguiente gramática, genera:

- La tabla de análisis predictivo. Emplea el algoritmo 4.31.
- ¿Se reconoce la entrada **bda**? Emplea el algoritmo 4.34.
- ¿Se reconoce la entrada **dba**? Emplea el algoritmo 4.34.

$$
\begin{aligned}
\mathrm{S} &\rightarrow \mathrm{A}\ \mathrm{a} \\
\mathrm{A} &\rightarrow \mathrm{B}\ \mathrm{D} \\
\mathrm{B} &\rightarrow \mathrm{b} \mid \epsilon \\
\mathrm{D} &\rightarrow \mathrm{d} \mid \epsilon
\end{aligned}
$$

Modo pánico:

- Agrega todos los símbolos del $FOLLOW(A)$ en el conjunto de sincronización de A.
- Utiliza la estructura jerárquica de las construcciones.
- Agrega todos los símbolo del $FIRST(A)$ a los conjuntos de sincronización del no terminal A.
- Si un no terminal genera un ϵ, entonces la producción que deriva ϵ es tomada por omisión.
- Si hay un terminal en el tope de la pila, se hace pop de ese terminal.

NON - TERMINAL	INPUT SYMBOL					
	id	+	*	(	)	$
E	$E \to TE'$			$E \to TE'$	synch	synch
E'		$E \to +TE'$			$E \to \epsilon$	$E \to \epsilon$
T	$T \to FT'$	synch		$T \to FT'$	synch	synch
T'		$T' \to \epsilon$	$T' \to *FT'$		$T' \to \epsilon$	$T' \to \epsilon$
F	$F \to$ **id**	synch	synch	$F \to (E)$	synch	synch

Figure 4.22: Synchronizing tokens added to the parsing table of Fig. 4.17

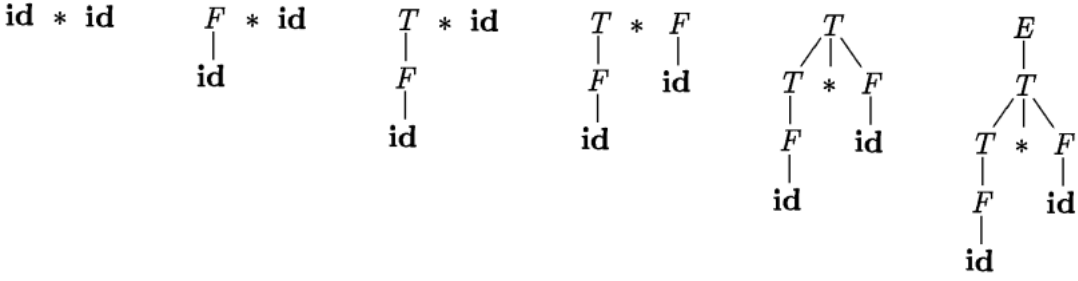

Figure 4.25: A bottom-up parse for **id** $*$ **id**

RIGHT SENTENTIAL FORM	HANDLE	REDUCING PRODUCTION
$\mathbf{id}_1 * \mathbf{id}_2$	$\mathbf{id}_1$	$F \to \mathbf{id}$
$F * \mathbf{id}_2$	F	$T \to F$
$T * \mathbf{id}_2$	$\mathbf{id}_2$	$F \to \mathbf{id}$
$T * F$	$T * F$	$E \to T * F$

Figure 4.26: Handles during a parse of $\mathbf{id}_1 * \mathbf{id}_2$

STACK	INPUT	ACTION
$	$\mathbf{id}_1 * \mathbf{id}_2$ $	shift
$ $\mathbf{id}_1$	$* \mathbf{id}_2$ $	reduce by $F \to \mathbf{id}$
$ F	$* \mathbf{id}_2$ $	reduce by $T \to F$
$ T	$* \mathbf{id}_2$ $	shift
$ $T *$	$\mathbf{id}_2$ $	shift
$ $T * \mathbf{id}_2$	$	reduce by $F \to \mathbf{id}$
$ $T * F$	$	reduce by $T \to T * F$
$ T	$	reduce by $E \to T$
$ E	$	accept

Figure 4.28: Configurations of a shift-reduce parser on input $\mathbf{id}_1 * \mathbf{id}_2$

Mientras que las operaciones principales son shift y reduce, en realidad hay 4 posibles acciones que debe realizar una analizador shift-reduce:

- Shift. Mueve el siguiente símbolo de la entrada al tope de la pila.
- Reduce. El lado derecho de la cadena a ser reducida debe estar en el tope de la pila. Localizar el lado izquierdo de la cadena dentro del stack y decidir con qué no terminar se reemplaza la cadena.
- Aceptar. Anuncia una terminación exitosa del análisis.
- Error. Descubre un error de sintaxis y llama a una rutina de recuperación de error.

Conflictos durante el análisis shift-reduce

Existen gramáticas libres de contexto para las cuales este tipo de analizadores no puede ser usado. Cada analizador shift-reduce puede alcanzar una configuración en la que el analizador, conociendo el contenido de la pila y el siguiente símbolo de entrada no puede decidir si desea reducir o cambiar (conflicto shift-reduce), o no puede decidir cual reducir aplicar (conflicto reduce-shift).

- El tipo más prevalente de analizadores bottom-up son llamados LR(k).
- Entre los analizadores de este tipo, revisaremos los Simple LR (SLR), los canonical-LR y los LALR. Estos últimos son los más empleados en las herramientas de generadores de analizadores.

¿Por qué analizadores LR?

- Los analizadores LR pueden ser construidos para reconocer virtualmente todos los lenguajes de programación para los cuales una gramática libre de contexto puede ser escrita.
- El método de análisis LR es el método de análisis shift-reduce sin retroceso más general conocido.
- Un analizador LR puede detectar un error sintáctico tan pronto como sea posible hacerlo en un escaneo de izquierda a derecha de la entrada.
- Las gramáticas que pude ser analizadas usando métodos LR son es un superconjunto propio de la clase de gramáticas que pueden ser analizadas con un método predictivo o LL.

Pero... construir un analizador LR para una gramática típica de un lenguaje de programación requiere de mucho trabajo para ser construido a mano. Afortunadamente, existen muchas herramientas para eso.

Un analizador LR realiza las decisiones shift-reduce manteniendo estados para determinar en dónde estamos en un análisis. Un *itemLR*(0) de una gramática G es una producción de G con un punto en alguna posición del cuerpo de la producción. Por ejemplo, la producción $A \to XYZ$ tiene cuatro items:

$$
\begin{aligned}
A &\to \cdot XYZ \\
A &\to X \cdot YZ \\
A &\to XY \cdot Z \\
A &\to XYZ \cdot
\end{aligned}
$$

Una colección de conjuntos de items LR(0), llamados la colección *canónica* LR(0), provee la base para la construcción de un autómata finito determinista que es usado para realizar decisiones de análisis. Este autómata es llamado *autómata LR(0)*.

SetOfItems CLOSURE(I) {
    $J = I$;
    repeat
        for (each item $A \to \alpha \cdot B\beta$ in J)
            for (each production $B \to \gamma$ of G)
                if ($B \to \cdot\gamma$ is not in J)
                    add $B \to \cdot\gamma$ to J;
    until no more items are added to J on one round;
    return J;
}

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

Figure 4.32: Computation of CLOSURE

```
void items(G') {
        C = CLOSURE({[S' → ·S]});
        repeat
                for ( each set of items I in C )
                        for ( each grammar symbol X )
                                if ( GOTO(I,X) is not empty and not in C )
                                        add GOTO(I,X) to C;
        until no new sets of items are added to C on a round;
}
```

Figure 4.33: Computation of the canonical collection of sets of LR(0) items

Autómata LR(0)

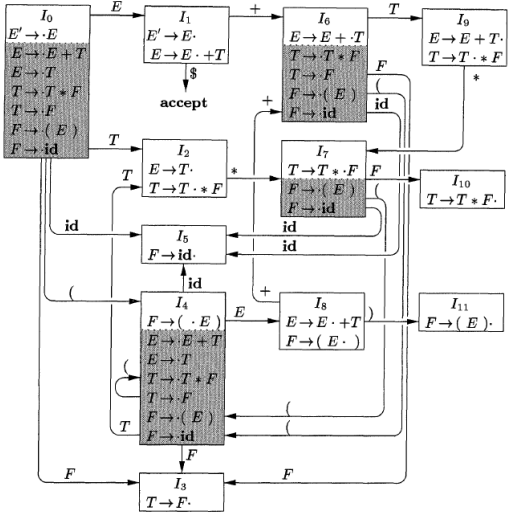

Figure 4.31: LR(0) automaton for the expression grammar (4.1)

Uso del autómata LR(0)

Si la cadena S de símbolos gramaticales hace que el autómata LR(0) de un estado inicial i a otro estado j, entonces cambiamos el siguiente símbolo a de la entrada. De lo contrario, elegimos reducir, el item en el estado j nos dirá que producción usar.

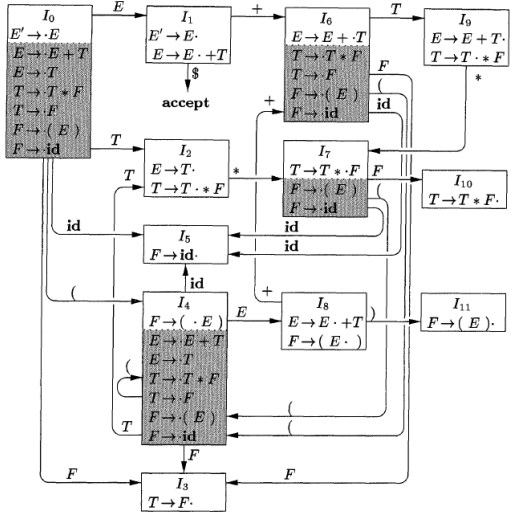

Figure 4.31: LR(0) automaton for the expression grammar (4.1)

id * id

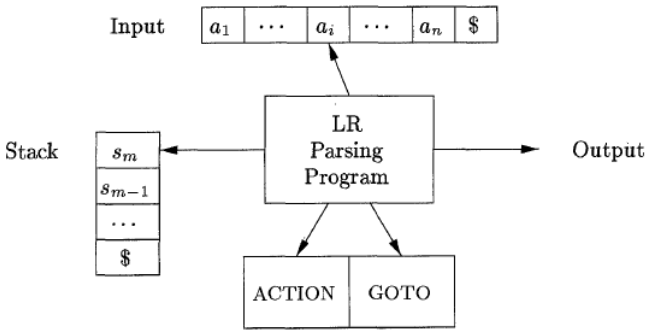

Figure 4.35: Model of an LR parser

Algorithm 4.46: Constructing an SLR-parsing table.

INPUT: An augmented grammar G'.

OUTPUT: The SLR-parsing table functions ACTION and GOTO for G'.

METHOD:

1. Construct $C = \{I_0, I_1, \ldots, I_n\}$, the collection of sets of LR(0) items for G'.

2. State i is constructed from I_i. The parsing actions for state i are determined as follows:

   (a) If $[A \to \alpha \cdot a\beta]$ is in I_i and $\text{GOTO}(I_i, a) = I_j$, then set $\text{ACTION}[i, a]$ to "shift j." Here a must be a terminal.

   (b) If $[A \to \alpha \cdot]$ is in I_i, then set $\text{ACTION}[i, a]$ to "reduce $A \to \alpha$" for all a in $\text{FOLLOW}(A)$; here A may not be S'.

   (c) If $[S' \to S \cdot]$ is in I_i, then set $\text{ACTION}[i, \$]$ to "accept."

   If any conflicting actions result from the above rules, we say the grammar is not SLR(1). The algorithm fails to produce a parser in this case.

3. The goto transitions for state i are constructed for all nonterminals A using the rule: If $\text{GOTO}(I_i, A) = I_j$, then $\text{GOTO}[i, A] = j$.

4. All entries not defined by rules (2) and (3) are made "error."

5. The initial state of the parser is the one constructed from the set of items containing $[S' \to \cdot S]$.

Si durante la construcción de la tabla existe algún conflicto con las reglas anteriores, entonces la gramática no es SLR.

STATE	ACTION						GOTO		
	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			

Figure 4.37: Parsing table for expression grammar

Generar el conjunto de elementos SLR, la función GOTO y la tabla de análisis de:

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

Ejercicio 4.6

Para la siguiente gramática:

$$S \to SS*$$
$$S \to SS+$$
$$S \to a$$

- Construye el conjunto de elementos SLR.
- Calcula las funciones $GOTO$.
- ¿Es una gramática SLR?

Algorithm 4.44: LR-parsing algorithm.

INPUT: An input string w and an LR-parsing table with functions ACTION and GOTO for a grammar G.

OUTPUT: If w is in $L(G)$, the reduction steps of a bottom-up parse for w; otherwise, an error indication.

METHOD: Initially, the parser has s_0 on its stack, where s_0 is the initial state, and $w\$$ in the input buffer. The parser then executes the program in Fig. 4.36. □

```
let a be the first symbol of w$;
while(1) { /* repeat forever */
        let s be the state on top of the stack;
        if ( ACTION[s, a] = shift t ) {
                push t onto the stack;
                let a be the next input symbol;
        } else if ( ACTION[s, a] = reduce A → β ) {
                pop |β| symbols off the stack;
                let state t now be on top of the stack;
                push GOTO[t, A] onto the stack;
                output the production A → β;
        } else if ( ACTION[s, a] = accept ) break; /* parsing is done */
        else call error-recovery routine;
}
```

Figure 4.36: LR-parsing program

STATE	ACTION						GOTO		
	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			

Figure 4.37: Parsing table for expression grammar

id * id + id

Ejercicio 4.7

Usando la tabla generada en el ejercicio anterior, muestra las acciones de análisis para la entrada $aa*a+$.

Analizadores LR más poderosos

Cómo se mencionó antes, existen dos métodos adicionales de análisis LR:

- "canonical-LR" (LR), el cual hace uso de los símbolos "lookahead". Este método usa un conjunto más grande de *items* llamado *items LR(1)*.
- "lookahead-LR" (LALR), el cual está basado en el conjunto de items LR(0), pero tiene menos estados que los de un analizador basado en items LR(1). Puede manejar más gramáticas que un SLR.

En el método SLR, cuando el estado *i* realizar una reducción por la producción $A \to \alpha$, si el conjunto de items I_i contiene $[A \to \alpha.]$ y *a* está en el *FOLLOW*(*A*). En algunas situaciones, sin embargo, cuando un estado *i* aparece en el tope de la pila, el prefijo viable $\beta\alpha$ está en la pila es tal qué $\beta\alpha$ no puede ser seguido por *a* en cualquier producción del lado derecho.
¿Recuerdas que sucedió con esta gramática?

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

- Es posible agregar más información al estado que nos permita determinar cuál de las reducciones aplicar a $A \rightarrow \alpha$. Esto se logra dividiendo los estados cuando sea necesario, haciendo que cada estado indique exactamente cuáles símbolos de entrada pueden seguir a α para el cuál hay una reducción de A.
- La información extra es incorporada dentro del estado redefiniendo los elementos para que tengan un segundo componente $[A \rightarrow \alpha.\beta, a]$, donde $A \rightarrow \alpha\beta$ es una producción y a es un terminal o el marcado \$. Llamamos a esto, un elemento $LR(1)$.

Construcción del conjunto de elementos LR(1)

SetOfItems CLOSURE(I) {
    repeat
        for (each item $[A \to \alpha \cdot B\beta, a]$ in I)
            for (each production $B \to \gamma$ in G')
                for (each terminal b in FIRST(βa))
                    add $[B \to \cdot\gamma, b]$ to set I;
    until no more items are added to I;
    return I;
}

SetOfItems GOTO(I, X) {
    initialize J to be the empty set;
    for (each item $[A \to \alpha \cdot X\beta, a]$ in I)
        add item $[A \to \alpha X \cdot \beta, a]$ to set J;
    return CLOSURE(J);
}

```
void items(G') {
    initialize C to CLOSURE({[S' → ·S, $]});
    repeat
        for ( each set of items I in C )
            for ( each grammar symbol X )
                if ( GOTO(I, X) is not empty and not in C )
                    add GOTO(I, X) to C;
    until no new sets of items are added to C;
}
```

Construir el conjunto de elementos LR(1) y la función GOTO para la siguiente gramática:

$$E \to E + T$$
$$E \to T$$
$$T \to T * F$$
$$T \to F$$
$$F \to (E) + T$$
$$F \to id$$

Construcción de tablas de análisis LR(1)

Algorithm 4.56: Construction of canonical-LR parsing tables.

INPUT: An augmented grammar G'.

OUTPUT: The canonical-LR parsing table functions ACTION and GOTO for G'.

METHOD:

1. Construct $C' = \{I_0, I_1, \cdots, I_n\}$, the collection of sets of LR(1) items for G'.
2. State i of the parser is constructed from I_i. The parsing action for state i is determined as follows.
   (a) If $[A \to \alpha \cdot a\beta, b]$ is in I_i and $\text{GOTO}(I_i, a) = I_j$, then set $\text{ACTION}[i, a]$ to "shift j." Here a must be a terminal.
   (b) If $[A \to \alpha \cdot, a]$ is in I_i, $A \neq S'$, then set $\text{ACTION}[i, a]$ to "reduce $A \to \alpha$."
   (c) If $[S' \to S \cdot, \$]$ is in I_i, then set $\text{ACTION}[i, \$]$ to "accept."

   If any conflicting actions result from the above rules, we say the grammar is not LR(1). The algorithm fails to produce a parser in this case.
3. The goto transitions for state i are constructed for all nonterminals A using the rule: If $\text{GOTO}(I_i, A) = I_j$, then $\text{GOTO}[i, A] = j$.
4. All entries not defined by rules (2) and (3) are made "error."
5. The initial state of the parser is the one constructed from the set of items containing $[S' \to \cdot S, \$]$.

Ejercicio 4.8

Construir el conjunto de elementos LR(1) y la función GOTO para la siguiente gramática:

$$E \to EE+$$
$$E \to EE*$$
$$E \to a$$

Construir la tabla de análisis.

Algorithm 4.59: An easy, but space-consuming LALR table construction.

INPUT: An augmented grammar G'.

OUTPUT: The LALR parsing-table functions ACTION and GOTO for G'.

METHOD:

1. Construct $C = \{I_0, I_1, \ldots, I_n\}$, the collection of sets of LR(1) items.

2. For each core present among the set of LR(1) items, find all sets having that core, and replace these sets by their union.

3. Let $C' = \{J_0, J_1, \ldots, J_m\}$ be the resulting sets of LR(1) items. The parsing actions for state i are constructed from J_i in the same manner as in Algorithm 4.56. If there is a parsing action conflict, the algorithm fails to produce a parser, and the grammar is said not to be LALR(1).

4. The GOTO table is constructed as follows. If J is the union of one or more sets of LR(1) items, that is, $J = I_1 \cap I_2 \cap \cdots \cap I_k$, then the cores of $\text{GOTO}(I_1, X)$, $\text{GOTO}(I_2, X), \ldots, \text{GOTO}(I_k, X)$ are the same, since $I_1, I_2, \ldots, I_k$ all have the same core. Let K be the union of all sets of items having the same core as $\text{GOTO}(I_1, X)$. Then $\text{GOTO}(J, X) = K$.

Construcción eficiente de tablas LALR

Existen varias modificaciones que podemos realizar al algoritmo anterior y así evitar la construcción completa de un conjunto LR(1):

- Primero, podemos representar cualquier conjunto LR(0) o LR(1) por su elemento inicial ($[S' \to S]$ or $[S' \to S, \$]$)
- Construimos el conjunto de kernels LALR(1) a partir del LR(0) usando un algoritmo de propagación y generación espontánea de lookaheads.
- Ya que tenemos los kernels LALR(1), podemos generar una tabla de análisis empleado el algoritmo que se presenta a continuación.

METHOD: The algorithm is given in Fig. 4.45. □

```
for ( each item A → α·β in K ) {
        J := CLOSURE({[A → α·β,#]} );
        if ( [B → γ·Xδ, a] is in J, and a is not # )
                conclude that lookahead a is generated spontaneously for item
                    B → γX·δ in GOTO(I, X);
        if ( [B → γ·Xδ, #] is in J )
                conclude that lookaheads propagate from A → α·β in I to
                    B → γX·δ in GOTO(I, X);
}
```

SetOfItems CLOSURE(I) {

repeat

for (each item $[A \to \alpha \cdot B\beta, a]$ in I)

for (each production $B \to \gamma$ in G')

for (each terminal b in FIRST(βa))

add $[B \to \cdot\gamma, b]$ to set I;

until no more items are added to I;

return I;

}

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

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_4: $L \to *\cdot R$

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: $S \to L = R\cdot$

METHOD:

1. Construct the kernels of the sets of LR(0) items for G. If space is not at a premium, the simplest way is to construct the LR(0) sets of items, as in Section 4.6.2, and then remove the nonkernel items. If space is severely constrained, we may wish instead to store only the kernel items for each set, and compute GOTO for a set of items I by first computing the closure of I.

2. Apply Algorithm 4.62 to the kernel of each set of LR(0) items and grammar symbol X to determine which lookaheads are spontaneously generated for kernel items in GOTO(I, X), and from which items in I lookaheads are propagated to kernel items in GOTO(I, X).

3. Initialize a table that gives, for each kernel item in each set of items, the associated lookaheads. Initially, each item has associated with it only those lookaheads that we determined in step (2) were generated spontaneously.

4. Make repeated passes over the kernel items in all sets. When we visit an item i, we look up the kernel items to which i propagates its lookaheads, using information tabulated in step (2). The current set of lookaheads for i is added to those already associated with each of the items to which i propagates its lookaheads. We continue making passes over the kernel items until no more new lookaheads are propagated.

FROM		TO	
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:	$S \to L = R\cdot$

Figure 4.46: Propagation of lookaheads

SET	ITEM	LOOKAHEADS			
		INIT	PASS 1	PASS 2	PASS 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_4:	$L \to *\cdot R$	$=$	$=$/\$	$=$/\$	$=$/\$
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:	$S \to L = R\cdot$				\$

Figure 4.47: Computation of lookaheads

Ejercicio 4.9

Empleando el algoritmo anterior, construye el conjunto de elementos LALR para la siguiente gramática:

$$S \rightarrow SS+$$
$$S \rightarrow SS*$$
$$S \rightarrow a$$

Ejercicio 4.10

Responde las siguientes preguntas:

- ¿Cuándo una gramática es LALR(1) pero no SLR(1)?
- ¿Cuándo una gramática es LR(1) pero no LALR(1)?

Revisar la sección 4.9 del libro de [AHO]