Autómatas de pilas y gramáticas libres de contexto

Pedro O. Pérez M., PhD.

Implementación de métodos computacionales
Tecnológico de Monterrey

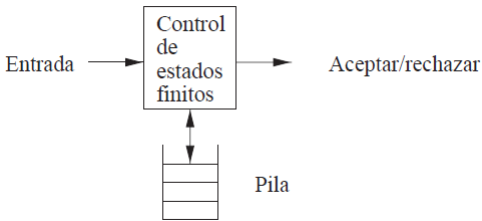
pperezm@tec.mx

06-2022

Contenido I

Definición de autómata de pila (AP)

- Existe un tipo de autómata que define loas lenguajes independientes de contexto. Dicho autómata, conocido como "autómata de pila" (PDA, pushdown automata), es una expresión del autómata finito no determinista con transiciones-ϵ, el cual constituye una forma de definir los lenguajes regulares. El autómata de pila es, fundamentalmente, un AFN-ϵ con la adición de una pila. La pila se puede leer,se pueden introducir elementos en ella y extraer sólo el elementos que está en la parte superior de la misma, exactamente igual que la estructura de datos "pila".
- La presencia de una pila significa que, a diferencia del autómata finito, el autómata de pila puede "recordar" una cantidad infinita de información.



El AP lee las entradas, un símbolo cada vez. El autómata puede observar el símbolo colocado en la parte superior de la pila y llevar a cabo su transición basándose en el estado actual, el símbolo de entrada y el símbolo que hay en la parte superior de la pila. Alternativamente, puede hacer una transición "espontánea", utilizando ϵ como entrada en lugar de un símbolo de entrada.

En una transición, el AP:

1. Consuma de la entrada el símbolo que se usa en la transición. Si como entrada se utiliza ϵ, entonces no se consume ningún símbolo de entrada.
2. Pasa a un nuevo estado, que puede o no ser el mismo que el estado anterior.
3. Reemplaza el símbolo de la parte superior de la pila por cualquier cadena.
   - Puede ser el mismo símbolo del tope de la pila (no hace nada).
   - Saca el tope de la pila (pop, corresponde a ϵ).
   - Cambia el primer elemento por otro (reemplazo sin *push* o *pop* o haciendo ambos).
   - Coloca un elemento en el tope de la pila posiblmente cambiando el primer elemento.

Definición formal de autómata de pila (AP)

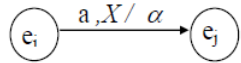
La notación formal de un AP incluye siete componentes. Escribirmos la especificación de un autómata de pila P de la siguiente forma: $P = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$.

El significado de cada uno de los componentes es el siguiente:

1. Q: Un conjunto finito de estados, como los estados de un autómata finito.
2. Σ: Un conjunto finito de *símbolos de entrada*, también análogo al componente correspondiente de un autómata finito.
3. Γ: Un *alfabeto de pila* finito. Este componente, que no tiene análogo en los autómatas finitos, es el conjunto de símbolo que pueden introducirse en la pila.

4. δ: La *función de transición*. Como en el autómata finito, δ controla el comportamiento del autómata. Formalmente, δ toma como argumento $\delta(q, a, X)$, donde q es un estado de Q, a es cualquier símbolo de entrada de Σ o ϵ y X es un símbolo de la pila que pertenece a Γ. La salida de δ es un conjunto finito de pares (p, γ), donde P es el nuevo estado y γ es la cadena de símbolos de la pila que reemplaza a X en la parte superior de la pila.
5. q_0: El *Estado inicial*. El autómata de pila se encuentra en este estado antes de realizar ninguna transición.
6. Z_0: El *Símbolo inicial*. Inicialmente, la pila del autómata consta de una instancia de este símbolo.
7. F: El conjunto de *estados de aceptación* o *estados finales*.

La función de transición de estados de un AP puede ser representada por un grafo donde los nodos representan los estados y los arcos transiciones.



Si el estado actual es e_i y la cabeza lectora apunta al símbilo a, y el tope de la pila es X, entonces cambia al estado e_j, avanza la cabeza lectura y sustituye el símbolo del tope, X, de la pila por la cadena α. Por ejemplo:

- Si $\alpha = ZYX$, deja X, agrega Y y Z (tope $= Z$), donde X, Y, $Z \in P$.
- Si $\alpha = XX$, deja X, agrega X (tope $= X$).
- Si $\alpha = X$, deja X como el mismo tope (no altera la pila).
- Si $\alpha = \epsilon$, elimina X, y el nuevo tope es el símbolo por debajo de la pila.

Ejemplo 1

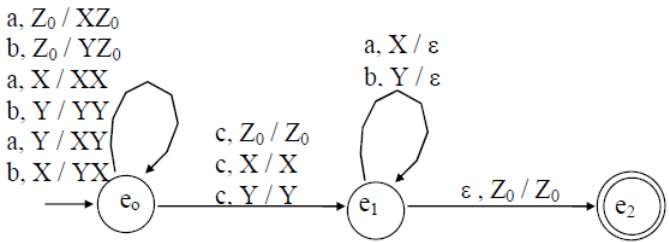
$L_1 = \{wcw^R \mid w \in \{a, b\}^*\}$

$APD_1 =< \{e_0, e_1, e_2\}, \{a, b, c\}, \{X, Y, Z_0\}, \delta, e_0, Z_0, \{e_2\} >$

δ

¿Cuál cadena es aceptada por APD_1?

- abcba
- c
- abcab
- a



Ejemplo 2

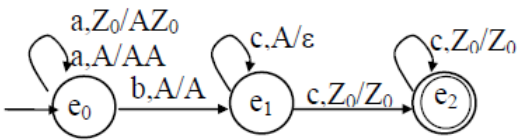
$L_2 = \{a^i bc^k \mid i, k \geq 1 \text{ y } i < k\}$

$APD_2 =< \{e_0, e_1, e_2\}, \{a, b, c\}, \{A, Z_0\}, \delta, e_0, Z_0, \{e_2\} >$

δ

¿Cuál cadena es aceptada por APD_2?

- aacbbb
- aacbb



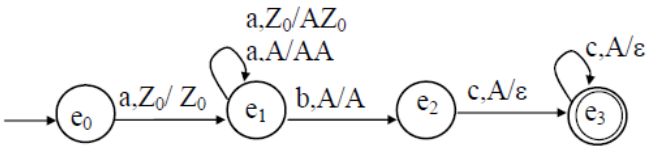
$L_3 = \{a^i bc^k \mid i, k \geq 1 \text{ y } i > k\}$

$APD_3 =< \{e_0, e_1, e_2, e_3\}, \{a, b, c\}, \{A, Z_0\}, \delta, e_0, Z_0, \{e_3\} >$

δ

¿Cuál cadena es aceptada por APD_1?

- aacbbb
- aacbb



Un ejemplo: analizador sintáctico

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

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.

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

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

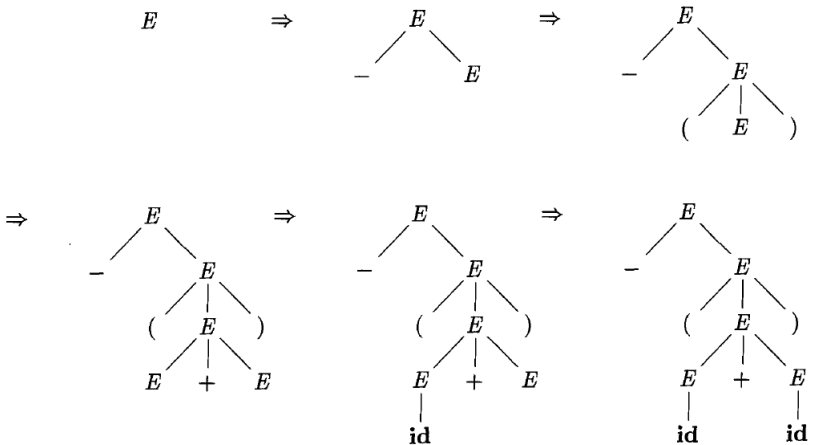


Figure 4.4: Sequence of parse trees for derivation (4.8)

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.

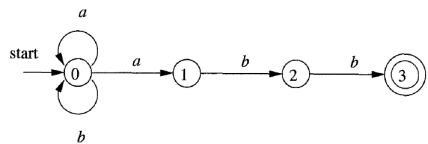


Figure 3.24: A nondeterministic finite automaton

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

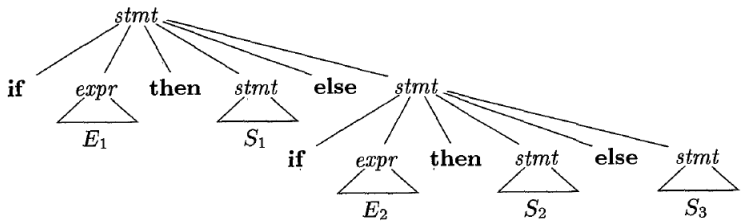


Figure 4.8: Parse tree for a conditional statement

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

has the two parse trees shown in Fig. 4.9.

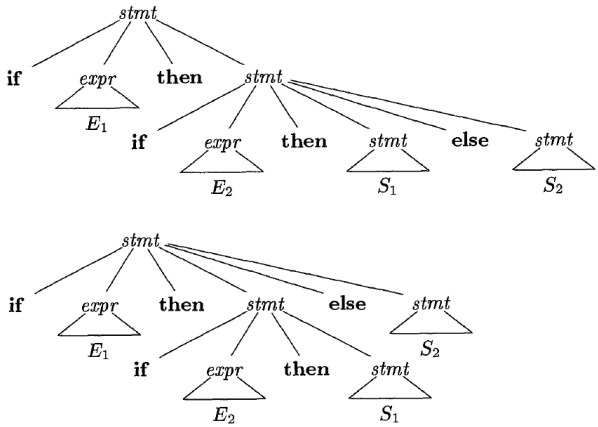


Figure 4.9: Two parse trees for an ambiguous sentence

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

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. $\square$

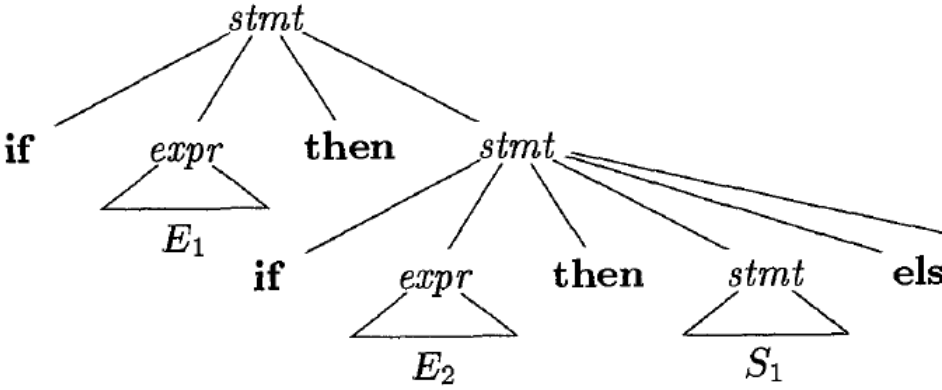
$S \rightarrow A\ a \mid b$
$A \rightarrow A\ c \mid S\ d \mid \epsilon$

```
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

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

has the two parse trees shown in Fig. 4.9.



$$A \rightarrow \beta A'$$
$$A' \rightarrow \alpha A' \quad | \quad \epsilon$$

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 código fuente.

- 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

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.

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 6

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.

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