Universidad Autónoma del Estado de Hidalgo

2.4 Análisis sintáctico

Ejercicios

ALUMNO: ROBERTO MARTÍNEZ MAGAÑA
Dr. Eduardo Cornejo-Velázquez

3.9. Ejercicios y Actividades

1. a) Escriba una gramática que genere el conjunto de cadenas

$s;, s;s;, s;s;s;, \ldots$.

SOLUCIÓN a)

$S \rightarrow s$
$S \rightarrow s\ S$

b) Genere un árbol sintáctico para la cadena $s;s;$

SOLUCIÓN b)

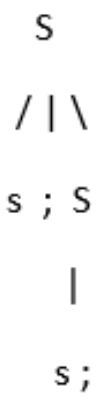

Figure 1: Ejercicio 1, b)

2. Considere la siguiente gramática:

$rexp \rightarrow rexp\ ''|''rexp$
$|rexprexp$
$|rexp''*''$
$|''(''rexp'')''$
$|letra$

a) Genere un árbol sintáctico para la expresión regular (ab—b)*.

Figure 2: Ejercicio 2)

3. De las siguientes gramáticas, describa el lenguaje generado por la gramática y genere árboles sintácticos con las respectivas cadenas.

a) $S \rightarrow s\ S + |SS * |a$ con la cadena $aa + a *$.

SOLUCIÓN a) Esta gramática genera expresiones postfijas con operandos a y operadores $+, *$.

Ejemplos de cadenas:

$aa + ß(a + a)$
$aa * ß(a * a)$
$aa + a * ß((a + a) * a)$

Figure 3: Ejercicio 3, a)

b) $S \rightarrow 0\ S1|01$ con la cadena 000111

SOLUCIÓN b) Genera cadenas balanceadas de ceros y unos con la forma: 0^n1^n con $n \geq 1$. (Ejemplo: 01, 0011, 000111, etc.)

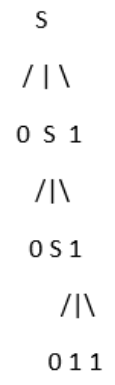

Figure 4: Ejercicio 3, b)

c) $S \rightarrow +\ SS|*SS|a$ con la cadena $+\ *a$aa.

SOLUCIÓN c) Genera expresiones prefijas con $+, *$ como operadores y a como operando.

Ejemplos:

$+aa \rightarrow (a + a)$
$aa \rightarrow (a * a)$
$+\ *aaa \rightarrow ((a * a) + a)$

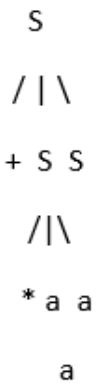

Figure 5: Ejercicio 3, c)

4. ¿Cuál es el lenguaje generado por la siguiente gramática?

$S \rightarrow xSy \mid e$

SOLUCIÓN: Esta gramática genera cadenas balanceadas de la forma: Cada x tiene un y correspondiente.

El lenguaje generado es:

$$L = \{x^n y^n \mid n \geq 0\}$$

Significa: todas las cadenas que tienen n `x` seguidos de n `y`, incluyendo la cadena vacía cuando $n = 0$. Siempre hay la misma cantidad de x y de y, y los x están antes que los y.

5. Genere el árbol sintáctico para la cadena *zazabzbz* utilizando la siguiente gramática:
$S \rightarrow zMNz$
$M \rightarrow aNa$
$N \rightarrow bNb$
$N \rightarrow z$

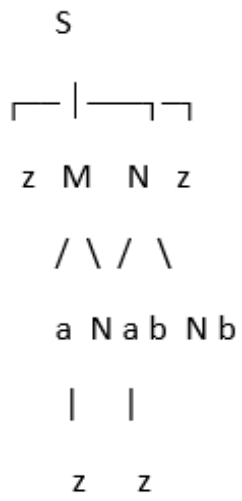

Figure 6: Ejercicio 5)

6. Demuestre que la gramática que se presenta a continuación es ambigua, mostrando que la cadena *ictictses* tiene derivaciones que producen distintos árboles de análisis sintáctico.

$S \rightarrow ictS$
$S \rightarrow ictSeS$
$S \rightarrow s$

SOLUCIÓN: Queremos demostrar que la cadena `ictictses` tiene más de una derivación.

*Primera derivación:

$S \to ictS$
$S \to ictS$
$S \to s$

Figure 7: Ejercicio 5)

*Segunda derivación:

$S \to ictSeS$
$S \to ictSeS$
$S \to s$
$S \to s$

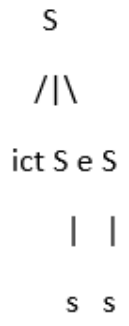

Figure 8: Ejercicio 5)

7. Considere la siguiente gramática

$S \to (L)|a$
$L \to L, S|S$

Encuéntrense árboles de análisis sintáctico para las siguientes frases:

$a)(a, a)$
$b)(a, (a, a))$
$c)(a, ((a, a), (a, a)))$

Figure 9: Ejercicio 7 a)

Figure 10: Ejercicio 7 b)

Figure 11: Ejercicio 7 c)

8. Constrúyase un árbol sintáctico para la frase *not* (true or false) y la gramática:

$bexpr \rightarrow bexprorbterm|bterm$
$bterm \rightarrow btermandbfactor|bfactor$
$bfactor \rightarrow notbfactor|(bexpr)|true|false$

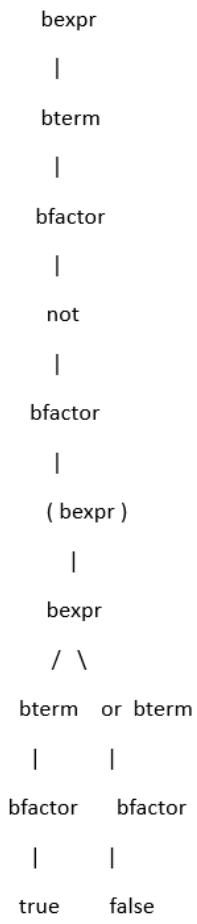

Figure 12: Ejercicio 8)

9. Diseñe una gramática para el lenguaje del conjunto de todas las cadenas de símbolos 0 y 1 tales que todo 0 va inmediatamente seguido de al menos un 1.

SOLUCIÓN: Podemos definir la siguiente gramática para este lenguaje:

$$S \to 1\,S \mid 0\,1\,S \mid \varepsilon$$

$S \to 1\,S$: Permite que la cadena comience con uno o más símbolos 1. Después de un 1, se puede seguir generando más cadenas válidas.

$S \to 0\,1\,S$: Garantiza que cualquier 0 esté seguido inmediatamente por un 1, y después permite seguir generando más cadenas válidas.

$S \to \varepsilon$: Permite la cadena vacía, que también es válida.

Algunos ejemplos de derivación:

Para la cadena 011, la derivación es:

$$S \to 0\,1\,S \to 0\,1\,1\,S \to 0\,1\,1\,\varepsilon \to 011$$

Para la cadena 111, la derivación es:

$$S \to 1\,S \to 1\,1\,S \to 1\,1\,1\,S \to 1\,1\,1\,\varepsilon \to 111$$

Para la cadena 0101, la derivación es:

$$S \to 0\,1\,S \to 0\,1\,0\,1\,S \to 0\,1\,0\,1\,\varepsilon \to 0101$$

10. Elimine la recursividad por la izquierda de la siguiente gramática:

$S \to (L)|a$
$L \to L, S|S$

SOLUCIÓN: La producción de L tiene recursividad por la izquierda en $L \to L, S$, ya que el símbolo L aparece en el lado izquierdo de su propia producción. Nuestro objetivo es transformar esta gramática para eliminar la recursividad por la izquierda.

Procedimiento para eliminar la recursividad por la izquierda

1. La producción recursiva $L \to L, S \mid S$ se puede reescribir como:

$$L \to SL'$$

$$L' \to, SL' \mid \varepsilon$$

Donde L' es un nuevo no terminal que representa la parte recursiva de la producción.

2. La producción de S no tiene recursividad por la izquierda, por lo que permanece igual:

$$S \to (L) \mid a$$

Gramática sin recursividad por la izquierda

La gramática transformada es la siguiente:

$$S \to (L) \mid a$$

$$L \rightarrow SL'$$

$$L' \rightarrow, SL' \mid \varepsilon$$

Explicación de la transformación

- La producción $S \rightarrow (L) \mid a$ se mantiene sin cambios, ya que no contiene recursividad por la izquierda.
- La producción $L \rightarrow SL'$ reemplaza la recursión por la izquierda, generando primero un S seguido de L'.
- La producción $L' \rightarrow, SL' \mid \varepsilon$ permite manejar las secuencias de comas seguidas de S, o finalizar con la cadena vacía ε.

Con esta transformación, hemos eliminado la recursividad por la izquierda de la gramática original, lo que la hace adecuada para ser procesada por algoritmos de análisis sintáctico que requieren gramáticas sin recursividad por la izquierda.

11. Dada la gramática $S \rightarrow (S)|x$, escriba un pseudocódigo para el análisis sintáctico de esta gramática mediante el método descendente recursivo.

SOLUCIÓN:

Función S():
Si el próximo símbolo es `'('`:
Consumir el símbolo `'('`
Llamar a la función `S()` para analizar el subárbol dentro de los paréntesis
Si el próximo símbolo es `')'`:
Consumir el símbolo `')'`
Aceptar la producción $S \rightarrow (S)$
Sino:
Reportar error: Se esperaba `')'`
Sino si el próximo símbolo es `'x'`:
Consumir el símbolo `'x'`
Aceptar la producción $S \rightarrow x$
Sino:
Reportar error: Se esperaba `'('` o `'x'`

12. Qué movimientos realiza un analizador sintáctico predictivo con la entrada $(id+id)*id$, mediante el algoritmo 3.2, y utilizándose la tabla de análisis sintáctico de la tabla 3.1. (Tómese como ejemplo la Figura 3.13).

Tabla 3.1
Tabla de análisis sintáctico para la gramática 3.3

No terminal	Símbolo de entrada					
	id	+	*	(	)	$
E	E → TE '			E → TE '		
E '		E ' → +TE '			E ' → ε	E ' → ε
T	T → FT '			T → FT '		
T '		T ' → ε	T ' → *FT '		T ' → ε	T ' → ε
F	*F* → id			*F* → (*E*)		

Figure 13: Tabla de análisis sintáctico de la tabla 3.1

Algoritmo 3.2: Análisis sintáctico predictivo, controlado por una tabla. (Aho, Lam, Sethi, & Ullman, 2008, pág. 226)

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

Salida: Si *w* está en el lenguaje de la gramática *L*(*G*), una derivación por la izquierda de *w*; de lo contrario, una indicación de error.

Método: Al principio, el analizador sintáctico se encuentra en una configuración con *w*$ en el búfer de entrada, y el símbolo inicial *S* de *G* en la parte superior de la pila, por encima de $.

establecer *ip* para que apunte al primer símbolo de *w*;
establecer *X* con el símbolo de la parte superior de la pila;
while (*X* ≠ $) { /* mientras la pila no está vacía*/
    if (*X* es *a*) extraer de la pila y avanzar *ip*; /* *a*=símbolo al que apunta *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* → $Y_1 Y_2 \ldots Y_k$) {
                enviar de salida la producción *X* → $Y_1 Y_2 \ldots Y_k$;
                extraer 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 cima de la pila;
}

Figure 14: Algoritmo 3.2

Pila	Entrada	Acción
$E	(id+id)*id$	E → TE'
$E'T	(id+id)*id$	T → FT'
$E'T'F	(id+id)*id$	F → (E)
$E'T')E(	(id+id)*id$	concuerda(()
$E'T')E	id+id)*id$	E → TE'
$E'T')E'T	id+id)*id$	T → FT'
$E'T')E'T'F	id+id)*id$	F → id
$E'T')E'T'id	id+id)*id$	concuerda(id)
$E'T')E'T'	+id)*id$	T' → ε
$E'T')E'	+id)*id$	E' → +TE'
$E'T')E'T+	+id)*id$	concuerda(+)
$E'T')E'T	id)*id$	T → FT'
$E'T')E'T'F	id)*id$	F → id
$E'T')E'T'id	id)*id$	concuerda(id)
$E'T')E'T'	)*id$	T' → ε
$E'T')E'	)*id$	E' → ε
$E'T')	)*id$	concuerda())
$E'T'	*id$	T' → *FT'
$E'T'F*	*id$	concuerda(*)
$E'T'F	id$	F → id
$E'T'id	id$	concuerda(id)
$E'T'	$	T' → ε
$E'	$	E' → ε
$	$	aceptar()

Figure 15: Tabla de movimientos

13. La gramática 3.2, sólo maneja las operaciones de suma y multiplicación, modifique esa gramática para que acepte, también, la resta y la división; Posteriormente, elimine la recursividad por la izquierda de la gramática completa y agregue la opción de que F, también pueda derivar en num, es decir, $F \rightarrow (E)|id|num$

Para incluir las operaciones de resta y división, se modifican las producciones de la siguiente manera:

$$E \rightarrow E + T \mid E - T \mid TT \rightarrow T * F \mid T/F \mid FF \rightarrow (E) \mid id \mid num$$

Para eliminar la recursividad por la izquierda, se introducen nuevos no terminales E' y T':

Para $E \rightarrow E + T \mid E - T \mid T$:

$$E \rightarrow TE'E' \rightarrow +TE' \mid -TE' \mid \epsilon$$

Para $T \to T * F \mid T/F \mid F$:

$$T \to FT'T' \to *FT' \mid /FT' \mid \epsilon$$

Para $F \to (E) \mid id \mid num$:

$$F \to (E) \mid id \mid num$$

La gramática final, sin recursividad por la izquierda y con las modificaciones anteriores, es:

$$E \to TE'E' \to +TE' \mid -TE' \mid \epsilon T \to FT'T' \to *FT' \mid /FT' \mid \epsilon F \to (E) \mid id \mid num$$

14. Escriba un pseudocódigo (e implemente en Java) utilizando el método descendente recursivo para la gramática resultante del ejercicio anterior (ejercicio 13).

```
funcion analizar():
    E()
    si entrada es $ entonces
        aceptar
    sino
        rechazar

funcion E():
    T()
    E'()

funcion E'():
    si entrada es + entonces
        consumir(+)
        T()
        E'()
    sino si entrada es - entonces
        consumir(-)
        T()
        E'()
    // epsilon

funcion T():
    F()
    T'()

funcion T'():
    si entrada es * entonces
        consumir(*)
        F()
        T'()
    sino si entrada es / entonces
        consumir(/)
```

```
--------F()
--------T'()
    // epsilon

funcion F():
    si entrada es ( entonces
        consumir(()
        E()
        consumir())
    sino si entrada es id entonces
        consumir(id)
    sino si entrada es num entonces
        consumir(num)
    sino
        rechazar
```

IMPLEMENTADO EN JAVA

```
import java.util.*;

public class Parser {
    private String input;
    private int position;
    private char currentToken;

    public Parser(String input) {
        this.input = input + "$"; // Agregar fin de cadena
        this.position = 0;
        this.currentToken = this.input.charAt(this.position);
    }

    private void consume(char expectedToken) {
        if (currentToken == expectedToken) {
            position++;
            if (position < input.length()) {
                currentToken = input.charAt(position);
            }
        } else {
            throw new RuntimeException("Error de sintaxis: se esperaba " + expectedToken);
        }
    }

    public void parse() {
        E();
        if (currentToken == '$') {
            System.out.println("An lisis completado con  xito .");
        } else {
            throw new RuntimeException("Error de sintaxis: entrada no consumida completame
        }
    }

    private void E() {
        T();
        EPrime();
```

```
}

private void EPrime() {
    if (currentToken == '+') {
        consume('+');
        T();
        EPrime();
    } else if (currentToken == '-') {
        consume('-');
        T();
        EPrime();
    }
    // epsilon
}

private void T() {
    F();
    TPrime();
}

private void TPrime() {
    if (currentToken == '*') {
        consume('*');
        F();
        TPrime();
    } else if (currentToken == '/') {
        consume('/');
        F();
        TPrime();
    }
    // epsilon
}

private void F() {
    if (currentToken == '(') {
        consume('(');
        E();
        consume(')');
    } else if (currentToken == 'i') {
        consume('i');
        consume('d');
    } else if (currentToken == 'n') {
        consume('n');
        consume('u');
        consume('m');
    } else {
        throw new RuntimeException("Error de sintaxis: token inesperado " + currentToke
    }
}

public static void main(String[] args) {
    Parser parser = new Parser("id+num*(id-num)");
    parser.parse();
}
```

}