

[illegible]

1. (A, b) es (b*S-, 3) porque el FIRST de la regla 3 de la gramatica ($A \rightarrow b * S -$) contiene b.
 2. (A, c) es (cdS, 2) porque el FIRST de la regla 2 de la gramatica ($A \rightarrow cdS$) contiene c.
 3. (S, e) es (e+, 4) porque el FIRST de la regla 4 de la gramatica ($S \rightarrow e +$) contiene e.
3. Muestre paso a paso el reconocimiento de la cadena: b*e+-#

Rem	Stack	Out	
b*e+-#	A#	ε	Comienza la regla 0 y estado inicial
b*e+-#	b*S-#	3	(A,b), agregamos 3 y sustituimos A
*e+-#	*S-#	3	pop b porque el top era b
e+-#	S-#	3	pop * porque el top era *
e+-#	e+-#	34	(S,e), agregamos 4 y sustituimos S
+ -#	+ -#	34	pop e porque el top era e
-#	-#	34	pop + porque el top era +
#	#	34	pop - porque el top era -
#	#	34	(accept)

2. Desarrolle un reconocedor flex y yacc para realizar el análisis sintáctico con esta gramática

```

alejandro@Mac ~/s/l/DanielOsoñoiox(main)>tb/out/mainmm
b*e+-#
194 // Para cualquier otro no termi
Linea reconocida correctamente antes por explorar
alejandro@Mac ~/s/l/DanielOsoñoioN(main)>na/out/main =
bbbbbb // Lo eliminamos de la pila
Error: syntaxerror

```

- **Estructura del entregable:**

- └─ src (carpeta con el codigo)
 - └─ grammar.y (archivo de bison)
 - └─ lexer.c (codigo generado de flex)
 - └─ lexer.l (archivo de flex)
 - └─ main.c (contiene la funcion main)
 - └─ parser.c (codigo fuente generado por bison)
 - └─ parser.h (encabezado generado por bison)

- **Compilar con make:**

Le adjunto un Makefile que realiza los pasos:

```

mkdir -p ./out
bison -ld src/grammar.y
flex -L src/lexer.l
mv grammar.tab.c src/parser.c
mv grammar.tab.h src/parser.h

```

```
mv lex.yy.c src/lexer.c
cc src/parser.c src/lexer.c src/main.c -o ./out/main
```

Para usarlo solo se debe usar:

```
make build
```

Y despues se puede correr el ejecutable con:

```
./out/main entrada.txt
```

- **Compilacion a mano:**

- **Generar bison y flex:**

```
bison -ld src/grammar.y
flex -L src/lexer.l
```

Los archivos generados se deben mover a la carpeta src con el nombre indicado:

```
mv grammar.tab.c src/parser.c
mv grammar.tab.h src/parser.h
mv lex.yy.c src/lexer.c
```

- **Compilar:**

Una vez generados los archivos de bison y flex (que deben ir colocados en la carpeta src con el nombre indicado arriba), podemos usar nuestro compilador de c de confianza:

```
cc src/parser.c src/lexer.c src/main.c -o ./out/main
```

3. Muestre por el método de fuerza bruta el reconocimiento de la cadena $b^*e+-\#$. Utilice la representación de cuatro elementos (estado, puntero a la cadena, cadena demostrada, cadena por demostrar) vista en clases.

1. $(Q, 1, \varepsilon, G\#)$
2. (Caso 1) $\vdash (Q, 1, G_1, A\#\#)$
3. (Caso 1) $\vdash (Q, 1, G_1 A_1, +b\#\#)$
4. (Caso 4) $\vdash (B, 1, G_1 A_1, +b\#\#)$
5. (Caso 6a) $\vdash (Q, 1, G_1 A_2, cdS\#\#)$
6. (Caso 4) $\vdash (B, 1, G_1 A_2, cdS\#\#)$
7. (Caso 6a) $\vdash (Q, 1, G_1 A_3, b * S - \#\#)$
8. (Caso 2) $\vdash (Q, 2, G_1 A_3 b, * S - \#\#)$
9. (Caso 2) $\vdash (Q, 3, G_1 A_3 b *, S - \#\#)$
10. (Caso 1) $\vdash (Q, 3, G_1 A_3 b * S_1, e + - \#\#)$
11. (Caso 2) $\vdash (Q, 4, G_1 A_3 b * S_1 e, + - \#\#)$
12. (Caso 2) $\vdash (Q, 5, G_1 A_3 b * S_1 e +, - \#\#)$
13. (Caso 2) $\vdash (Q, 6, G_1 A_3 b * S_1 e + -, \#\#)$
14. (Caso 2) $\vdash (Q, 7, G_1 A_3 b * S_1 e + - \#, \#)$
15. (Caso 3) $\vdash (T, 7, G_1 A_3 b * S_1 e + - \#, \varepsilon)$

Explicación, cada número corresponde con un paso del reconocimiento:

1. Inicia con el estado en recorrido hacia delante (Q), en el primer caracter, habiendo aplicado ε reglas, con la regla de entrada por verificar.
 2. Como lo que hay por verificar es un no terminal, entonces lo expandimos por su primera producción, es decir el caso 1.
 3. La expansión nos deja con $A\#$, el siguiente elemento a verificar también es no terminal (A), por lo que de nuevo aplica el caso 1 y lo expandimos.
 4. La expansión deja arriba de la pila sent al terminal +, que no es igual al primer caracter de la cadena de entrada, por lo que iniciaremos el backtrack, ponemos el estado en B .
 5. Como estamos en backtrack, y encontramos a A_1 como la última producción con la que contabamos, y sabemos que quedan más variantes de A , cambiamos de A_1 a A_2 y colocamos su rhs en la pila sent
 6. La variante A_2 nos deja con c como siguiente elemento, que no hace match con el primer elemento de la cadena $b^*e+-\#$, por lo que de nuevo ponemos el estado en B
 7. Remplazamos A_2 por la siguiente variante de A (A_3), recordemos que en fuerza pruta probamos todas las posibilidades.
 8. Hacemos match de b con el primer caracter de $b^*e+-\#$. Aumentamos el numero del caracter por matchear del 1 a 2.
 9. Hacemos match de * con el segundo caracter de $b^*e+-\#$. Aumentamos el numero del caracter por matchear del 2 a 3.
 10. Queda S en sent, así que lo expandimos por su primera variante $S - 1$
 11. Hacemos match de e con el tercer caracter de $b^*e+-\#$. Aumentamos el numero del caracter por matchear del 3 a 4.
 12. Hacemos match de + con el cuarto caracter de $b^*e+-\#$. Aumentamos el numero del caracter por matchear del 4 a 5.
 13. Hacemos match de - con el quinto caracter de $b^*e+-\#$. Aumentamos el numero del caracter por matchear del 5 a 6.
 14. Hacemos match de # con el sexto caracter de $b^*e+-\#$. Aumentamos el numero del caracter por matchear del 6 a 7.
 15. Como solo queda # en sent, y el numero de caracteres ya identificados es mayor a la longitud de la cadena de entrada, marcamos el estado con T , se reconocio la cadena exitosamente.
4. En el algoritmo de fuerza bruta cómo modificaría usted el código para aceptar solamente las reglas de orden par de un no terminal? (20%)

Se puede lograr esto por distintas maneras:

- Al momento de parsear la gramática del archivo de entrada descartar las variantes de cada no terminal que no sean pares, manteniendo un contador del numero de variante para este motivo.
- Al momento de expandir las producciones en el caso “1” en lugar de expandir la primera producción, poner directamente la segunda producción, y para cada caso de sustitución debido al backtrack (caso “6a”) en lugar de pasar de $A_i \rightarrow A_{i+1}$ hacer $A_i \rightarrow A_{i+2}$

El caso mas relacionado con el pseudocodigo es el segundo, por lo que procedo a detallar la respuesta:

- Para el caso 1:

```
State::Q => match self.sent.last().copied().unwrap() {
  // Si es un no terminal entonces lo expandimos por su primera
  // produccion
  Element::NonTerminal(id) => {
    self.caso = "1";
    self.pop_with(self.grammar.non_terminal[id]);
    self.extend_with(self.grammar.rhs[self.grammar.starting_ptr[id]]);
    self.hist.push((Element::NonTerminal(id), 0));
  }
  _ => { /* Otros casos */ }
}
```

pasa a ser:

```
State::Q => match self.sent.last().copied().unwrap() {
  // Verificamos si tienes mas de una regla (o sea una 2da regla, por lo menos)
  Element::NonTerminal(id) if self.grammar.number_rules[id] > 1 => {
    self.caso = "1";
    self.pop_with(self.grammar.non_terminal[id]);
    // Aplicamos la segunda variante, es decir,
    // al indice de inicio de variantes del no terminal le sumamos 1
    self.extend_with(self.grammar.rhs[self.grammar.starting_ptr[id] + 1]);
    /*
      Ya no se trata de la primera produccion (0 + 1) , sino que
      de la segunda (1 + 1)
    */
    self.hist.push((Element::NonTerminal(id), 1));
  }
  Element::NonTerminal(_) => { /* No hacemos nada */ }
  _ => { /* Otros casos */ }
}
```

- Para el caso 6a:

```
State::Q => match self.sent.last().copied().unwrap() {
  (Element::NonTerminal(id), _) if self.rem_variants(id) > 0 => {
    self.caso = "6a";
    self.state = State::Q;

    let n = unsafe { self.increase_last_hist_counter() };
    let start = self.grammar.starting_ptr[id];

    self.pop_with(self.grammar.rhs[start + n - 1]);
    self.extend_with(self.grammar.rhs[start + n]);
  }
}
```

pasa a:

```
State::Q => match self.sent.last().copied().unwrap() {
  /*
    Ahora no solo nos interesa que tenga mas de 0 variantes, sino que
    tenga por lo menos 2 para poder saltar de par en par
  */
  (Element::NonTerminal(id), _) if self.rem_variants(id) > 1 => {
    self.caso = "6a";
    self.state = State::Q;
  }
}
```

```

let n = unsafe { self.increase_last_hist_counter() };
let start = self.grammar.starting_ptr[id];

// Entonces sustituimos la variante en sent por la nueva
// variante a probar, como ahora es por pares disminuimos n en 2
self.pop_with(self.grammar.rhs[start + n - 2]);
self.extend_with(self.grammar.rhs[start + n]);
}
}

```

lo que requiere a su vez que modifiquemos `increase_last_hist_counter()`:

```

unsafe fn increase_last_hist_counter(&mut self) -> usize {
    self.hist
        .last_mut()
        .map(|(_, n)| {
            /* Aumentamos el contador de hist en 2 en lugar de 1 */
            *n += 2;
            *n
        })
        .unwrap_unchecked()
}

```

donde ahora realiza `*n += 2` en lugar de `*n += 1`