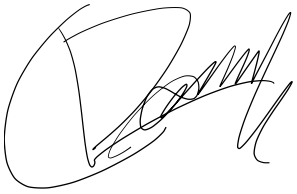
Universidad Panamericana
Campus México

Facultad de Ingeniería

Materia: Teoría de lenguajes y programación
Examen: Segundo Parcial
Profesor: Félix O Martínez Ríos
Fecha: 30/Octubre/2019

Nombre: Daniel Alejandro Osornio López. Carrera: Ing. Ineligencia de Datos & Ciberseguridad

Temario C



Dada la siguiente gramática:

1. G → A#
2. A → +b
3. A → cdS
4. A → b*S-
5. S → e+
6. S → -f

a. Demuestre que es una gramática LL(1) sin reglas vacías. Construya la tabla de reconocimiento y explique al menos tres entradas de la tabla. Muestre paso a paso el reconocimiento de la cadena: **b*e+-#**. (20%)
b. Desarrolle un reconocedor flex y yacc para realizar el análisis sintáctico con esta gramática (30%)
c. 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. (30%)
d. En el algoritmo de fuerza bruta cómo modificaría usted el código para aceptar solamente las reglas de orden par de un no terminal? (20%)

Respuestas

1. Dada la siguiente gramática

   $G \to A\#$
   $A \to +b \mid cdS \mid b * S -$
   $S \to e + \mid - f$

   1. Demuestre que es una gramática LL(1) sin reglas vacías.

      Para ser una gramatica LL(1) sin reglas vacias, la gramatica no debe tener reglas vacías y debe ser independiente del contexto.

      - No hay ninguna regla de la forma $A \to \varepsilon$, es decir, que no hay reglas vacías.
      - Es independiente del contexto pues para todas las expresiones de la forma $\alpha \to \beta$, α contiene solamene no terminales.
      - No hay intersecciones entre los conjuntos `FIRST` de las reglas alternativas para cada no terminal:

        1. Para G, no verificamos pues tiene una sola variante
        2. Para: A

           $\text{FIRST}(+b) \cap \text{FIRST}(cdS) = \{+\} \cap \{c\} = \emptyset$
           $\text{FIRST}(+b) \cap \text{FIRST}(b * S -) = \{+\} \cap \{b\} = \emptyset$
           $\text{FIRST}(cdS) \cap \text{FIRST}(b * S -) = \{c\} \cap \{b\} = \emptyset$

        3. Para: S

           $\text{FIRST}(e +) \cap \text{FIRST}(-f) = \{e\} \cap \{-\} = \emptyset$

      > Nota: Si eliminaramos la regla 0 de la gramática, que es redundante porque solo agrega de forma explicita el caracter de terminación, nos damos cuenta que se trata de una gramatica LL(1) simple porque todas las reglas son de la forma $A \to a\alpha$, donde cada variante de cada no terminal tiene al inicio un terminal distinto.

   2. Construya la tabla de reconocimiento y explique al menos tres entradas de la tabla.

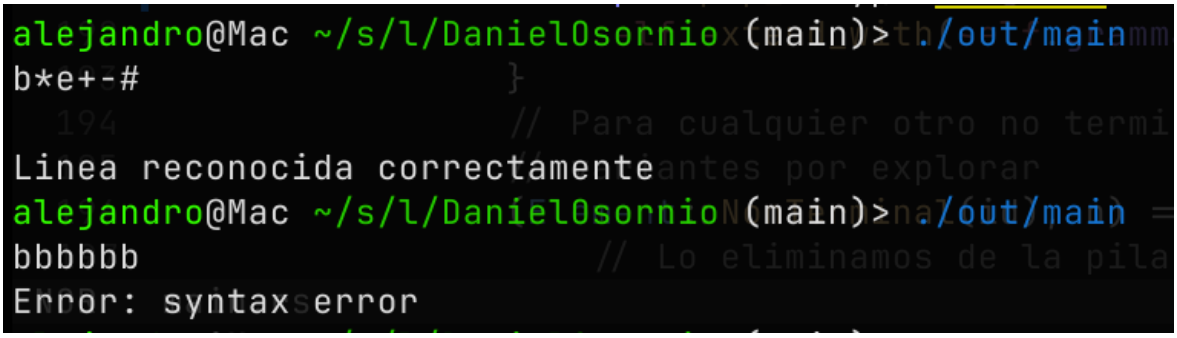
	b	c	d	e	f	+	*	-	#
A	(b*S-, 3)	(cdS, 2)					(+b, 1)		
S				(e+, 4)				(-f, 5)	
b	pop								
c		pop							
d			pop						
e				pop					
f					pop				
+						pop			
*							pop		
-								pop	
#									accept

1. (A, b) es (b*S-, 3) porque el `FIRST` de la regla 3 de la gramatica ($A \to b * S -$) contiene b.
2. (A, c) es (cdS, 2) porque el `FIRST` de la regla 2 de la gramatica ($A \to cdS$) contiene c.
3. (S, e) es (e+, 4) porque el `FIRST` de la regla 4 de la gramatica ($S \to e +$) 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



- **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.

$$
\begin{aligned}
&1.(Q, 1, \varepsilon, G\#) \\
&2.(\text{Caso } 1) && \vdash (Q, 1, G_1, A\#\#) \\
&3.(\text{Caso } 1) && \vdash (Q, 1, G_1A_1, +b\#\#) \\
&4.(\text{Caso } 4) && \vdash (B, 1, G_1A_1, +b\#\#) \\
&5.(\text{Caso } 6a) && \vdash (Q, 1, G_1A_2, cdS\#\#) \\
&6.(\text{Caso } 4) && \vdash (B, 1, G_1A_2, cdS\#\#) \\
&7.(\text{Caso } 6a) && \vdash (Q, 1, G_1A_3, b*S-\#\#) \\
&8.(\text{Caso } 2) && \vdash (Q, 2, G_1A_3b, *S-\#\#) \\
&9.(\text{Caso } 2) && \vdash (Q, 3, G_1A_3b*, S-\#\#) \\
&10.(\text{Caso } 1) && \vdash (Q, 3, G_1A_3b*S_1, e+-\#\#) \\
&11.(\text{Caso } 2) && \vdash (Q, 4, G_1A_3b*S_1e, +-\#\#) \\
&12.(\text{Caso } 2) && \vdash (Q, 5, G_1A_3b*S_1e+, -\#\#) \\
&13.(\text{Caso } 2) && \vdash (Q, 6, G_1A_3b*S_1e+-, \#\#) \\
&14.(\text{Caso } 2) && \vdash (Q, 7, G_1A_3b*S_1e+-\#, \#) \\
&15.(\text{Caso } 3) && \vdash (T, 7, G_1A_3b*S_1e+-\#, \varepsilon)
\end{aligned}
$$

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`, asi 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 $A_i \to A_{i+1}$ hacer $A_i \to A_{i+2}$

   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 */ }i
    _ => { /* 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`