
# Universidad Nacional Autónoma de México

## Facultad de Ciencias

### Compiladores

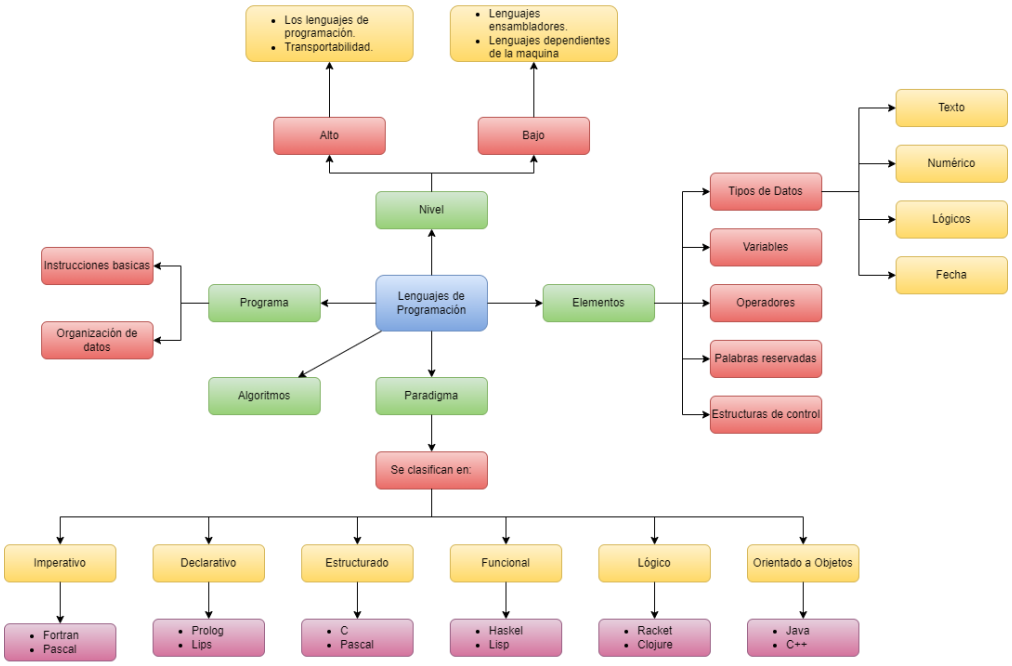
Tarea 1

Autores:

Escamilla Soto Cristopher Alejandro
Montiel Manriquez Ricardo

3 de Marzo de 2022

1. [**1.5pts**] Realiza un mapa mental para clasificar los lenguajes de programación de la forma en que te parezca mejor, incluye ejemplos de lenguajes y explica las características que consideras para esta clasificación.



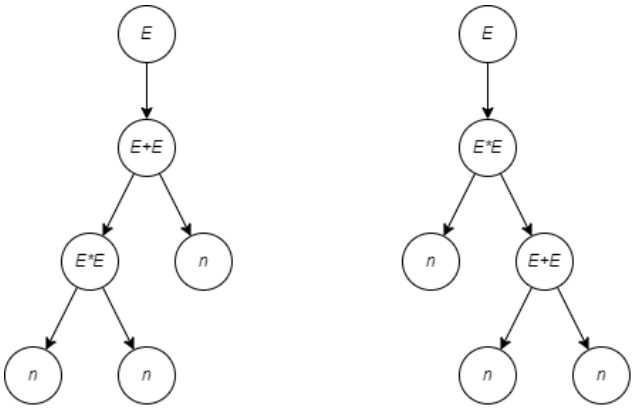

2. [**2pts**] Considera la siguiente gramática de expresiones aritméticas:

$$E \rightarrow \ E + E \mid E * E \mid n$$

- Esta gramática es ambigua. Da dos ejemplos de expresiones que tengan diferentes derivaciones para demostrarlo.

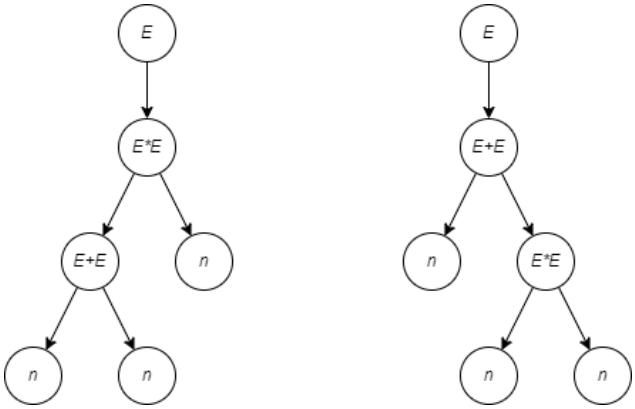
  **Ejemplo 1:**

  Tenemos la siguiente expresión: $n * n + n$ y podemos generar los siguientes dos arboles derivados por lo que es ambigua.




**Ejemplo 2:**

Tenemos la siguiente expresión: $n + n * n$ y podemos generar los siguientes dos arboles derivados por lo que es ambigua.




- Transforma la gramática en una que no sea ambigua y que genere el mismo lenguaje.

$$E \rightarrow \ E + F \ \mid \ F \ \mid \ n$$
$$F \rightarrow \ F * F \ \mid \ n$$

3. [**2pts**] La siguiente tabla define los tokens para un lenguaje simple donde

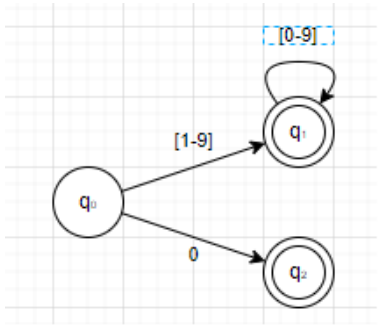
$$\sum = \{a, \ ... \, z \ , \ 0 \ ... \ 9 \ , \ . \ , \ \oplus \ , \ (, )\}$$

| *token* | *exp. regular* |
|---|---|
| **num** | $0 + [1-9][0-9]^*$ |
| **lam** | *lam* |
| **id** | $[a-z][a-z+0-9]^*$ |
| **dot** | . |
| **lp** | ( |
| **rp** | ) |
| **binop** | $\oplus$ |

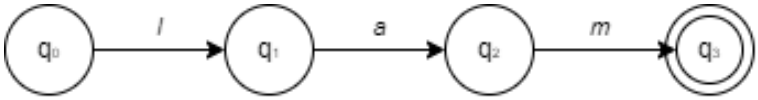
Construye un autómata finito determinista que acepte los las cadenas de este lenguaje. Puede usar algún método, eg. derivadas de expresiones regulares o construcción de un $AFN_\epsilon$ y transformaciones. Indica el método usado y mostrar el proceso.

Antes de hacer el $AFN_\epsilon$ debemos de crear el autómata para cada token y que este reconozca su lenguaje.

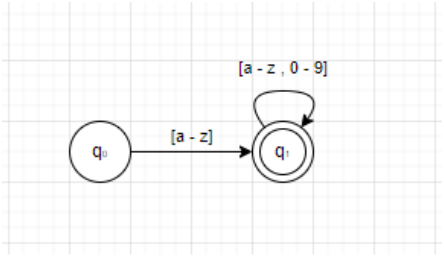
- Autómata del token num:



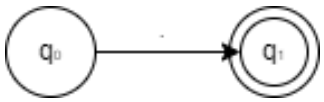

- Autómata del token lam:



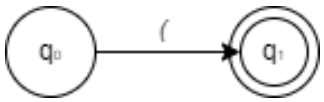

- Autómata del token id:



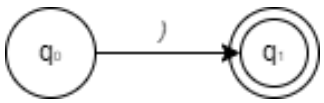
- Autómata del token dot:



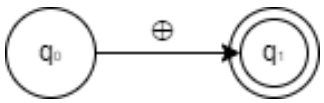

- Autómata del token lp:



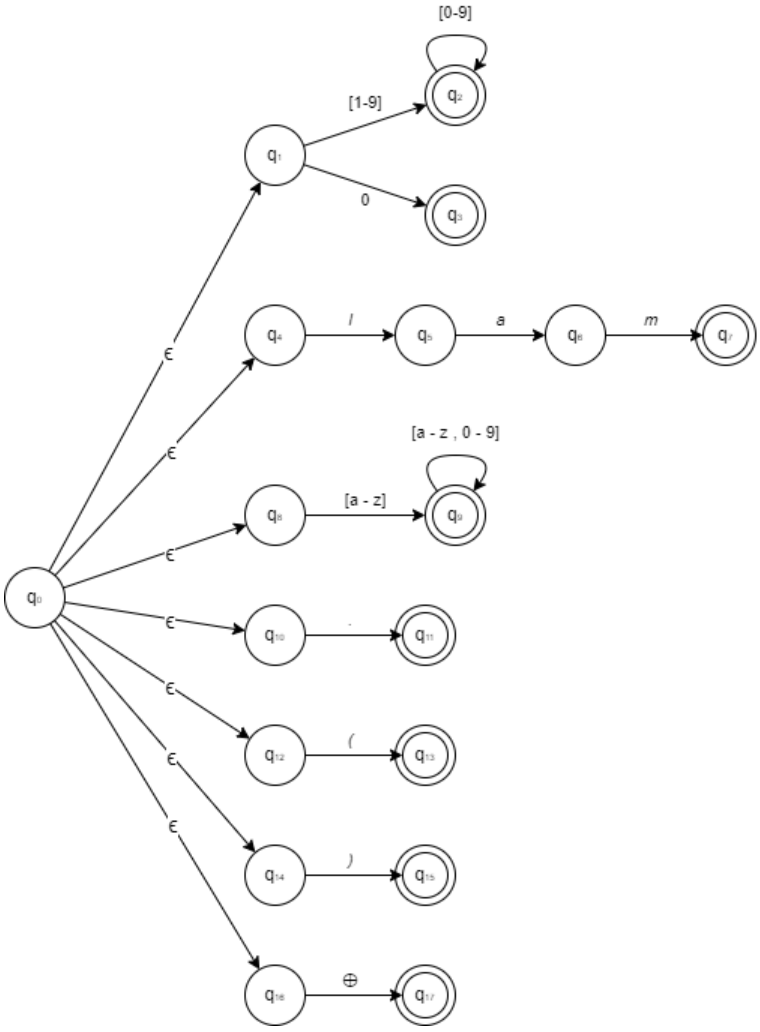

- Autómata del token rp:



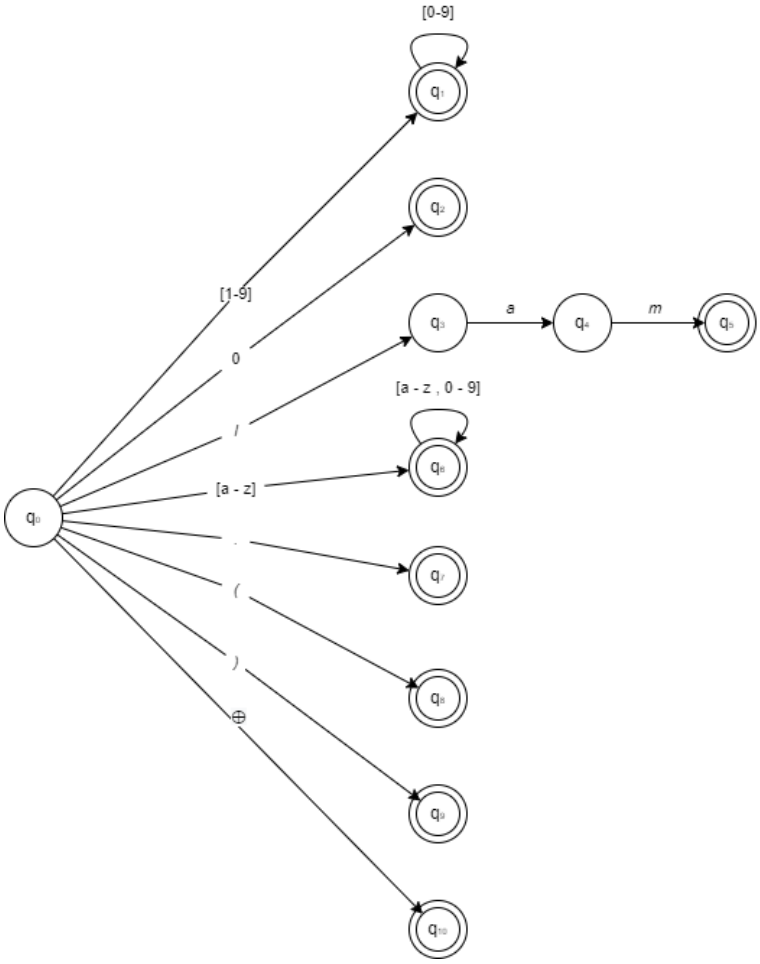

- Autómata del token Binop:




Ahora tenemos que unir todos estos autómatas en uno solo utilizando las épsilon transiciones.




Y una vez que quitamos las epsilon transiciones nos queda:




4. [**2.5pts**] Un comentario en lenguaje C está definido como la secuencia de caracteres que empieza por /*, seguido del comentario y terminando con */. El comentario puede contener los caracteres * y / pero no seguidos, aunque la única excepción es que aparezca justamente */.

   - Demuestra que la siguiente expresión regular no describe correctamente los comentarios en C.

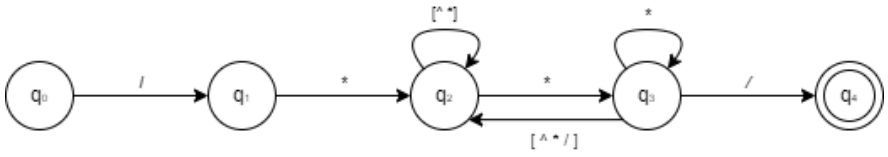
     $$\texttt{/* /* ([\^{} * /] | [\^{} *]/ | *[\^{} /])* *}^{\star}\texttt{ */}$$

     No funciona por que el lenguaje que nos dan no genera la cadena: $/**/*/$ donde el comentario seria la cadena * / divididos por un espacio, dicha cadena debería ser valida pero utilizando el lenguaje que nos dan, no la podríamos generar entonces proponemos el siguiente lenguaje:

     $$\texttt{/ * /* (( *}^{\star}\texttt{ [\^{} /] )}^{\star}\texttt{ + (( [\^{} * /])}^{\star}\texttt{ + (/ + } \epsilon \texttt{) *)*) *}^{\star}\texttt{ * /}$$

   - Construye un autómata finito determinista que acepta comentarios en C y obtén la expresión regular correcta (puedes usar algún método, eg. derivadas de expresiones

regulares o construcción de un **AFN** y transformaciones. Indicar el método usado y mostrar el proceso.)




Ten en cuenta que a* es la cerradura de Kleene, | es la unión de expresiones regulares y [^ a b c ] indica que puede incluirse cualquier caracter excepto los simbolos a, b y c.

5. **[2pts]** Considera la siguiente definición para un analizador léxico:

```
type token = ARR | LPAREN | RPAREN | FUN | VAR of string

rule token = parse
| whitespace+ { token lexbuf } (* skipwhitespace *)
| 'fun'       { FUN }
| character+  { VAR ( lexeme lexbuf ) }
| "->"        { ARR }
| '('         { LPAREN }
| ')'         { RPAREN }
| _ as c      { failwith "unexpected char" }
```

De acuerdo a la definición anterior, ¿cuántos tokens serán producidos al analizar las cadenas?

`fun x ->z x` y `(fun w ->w w)`

Da una tabla de los tokens y sus atributos para cada caso. La tabla puede ser de la siguiente forma:

| Lexema | Token | Valor |
|---|---|---|
| | | |

**fun x -> z x**

Se producen 5 tokens, un FUN para el fun, un VAR para x, un ARR para `->`, de nuevo VAR para z y otro VAR para x.

| Lexema | Token | Valor |
|---|---|---|
| fun | FUN | - |
| x | VAR | entrada en la tabla |
| -> | ARR | - |
| z | VAR | entrada en la tabla |
| x | VAR | entrada en la tabla |

**(fun w -> w w)**

Se producen 7 tokens, un LPAREN para (, un FUN para el fun, un VAR para w, un ARR para `->`, otro VAR para w, un ultimo VAR para w, y un RPAREN para ).

| Lexema | Token | Valor |
|---|---|---|
| ( | LPAREN | - |
| fun | FUN | - |
| w | VAR | entrada en la tabla |
| -> | ARR | - |
| w | VAR | entrada en la tabla |
| w | VAR | entrada en la tabla |
| ) | RPAREN | - |