



Universidad Nacional Autónoma de México

FACULTAD DE CIENCIAS

TAREA 01

Compiladores

Bonilla Reyes Dafne
Castañón Maldonado Carlos Emilio
García Ponce José Camilo
Villalpando Velázquez Diego Alfredo

Profesora: Lourdes del Carmen González Huesca

Ayudante: Fausto David Hernández Jasso
Ayudante: Juan Alfonso Garduño Solís
Ayudante: Juan Pablo Yamamoto Zazueta

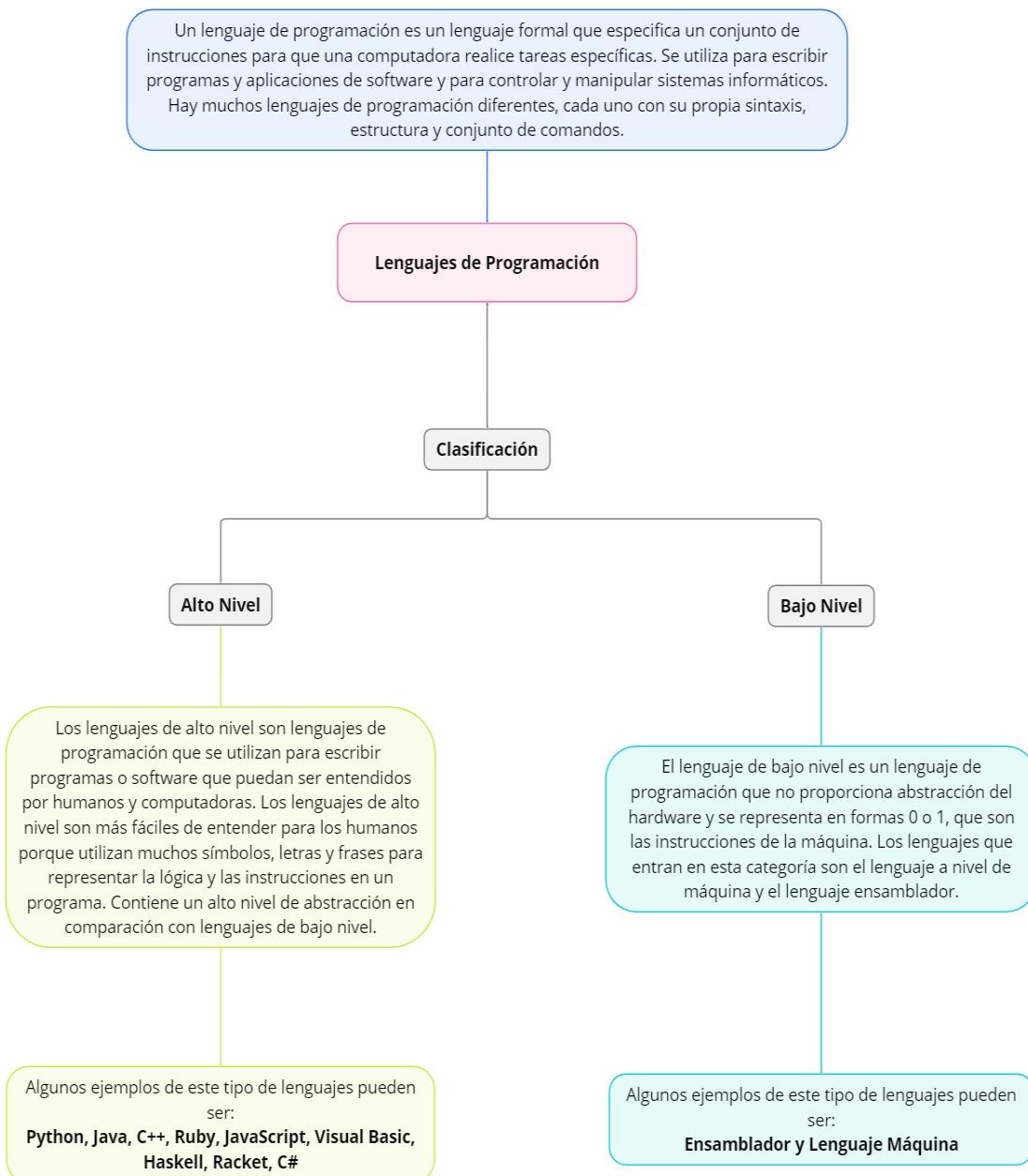
Febrero 2024



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 importantes para esta clasificación. No olvides incluir las referencias que consultes.

Siguiendo la forma de clasificar lenguajes de programación del libro de Programming Language Pragmatics, entonces clasificaremos a los lenguajes de programación por su nivel de abstracción, esto es:

- ★ Alto nivel
- ★ Bajo nivel





2. [2pts] Considera la siguiente expresión regular para reconocer números reales:

$$(+| -|\epsilon)digit^*.digitdigit^*(e(+| -|\epsilon)digitdigit^*|\epsilon)$$

Analiza esta expresión y explica qué problemática tiene para reconocer números reales. ¿Es posible arreglarla? ¿Cuál es una definición correcta?

Al analizar la presente podemos darnos cuenta de que el primer problema con el que nos encontramos es el de que al tener como primeros argumento a $(+| -|\epsilon)digit^*.digitdigit^*$ podríamos tener un numero como:

– .7

El cual no es un numero real, lo cual es grave ya que llegaríamos a casos como estos:

$$.00009 \neq 0.00009$$

Recordando que $\mathbb{Z} \subset \mathbb{R}$ podemos dar que una de las primeras modificaciones que tendremos será la de que en vez de tener $(+| -|\epsilon)$ tengamos $(-|\epsilon)$ ya que como buscamos la compatibilidad con los números reales, sabemos que estos no necesitan del signo + para indicarnos que son positivos.

Ahora, para arreglar el problema de tener números de la forma .8, cambiamos nuestro $digit^*$ a un $digit^+$, ya que con esto nos aseguraremos de que sea mínimo un dígito real (o mas) el que le tengamos que pasar, quedando de esta forma:

$$(-|\epsilon)digit^+$$

Luego, como estamos trabajando con reales ya tenemos arreglado la parte de los enteros pero no la de los números decimales, es por esto que retomamos a $(-|\epsilon)$ (siguiendo la lógica anterior), ya que tenemos nuestro signo observamos que necesitamos de el numero decimal que le acompañe, es por esto que aplicando nuestro truco de $digit^+$, tendremos por consiguiente a $digit^+.digit^+$ lo cual nos asegura que sin importar cuantos dígitos sean, siempre existirá un dígito en la parte izquierda o derecha del punto decimal, quedando de esta forma lo anterior como:

$$(-|\epsilon)digit^+.digit^+$$

Ya que tenemos nuestra forma de representar números enteros y números decimales, solo hace falta juntarlos con un OR, o en otras palabras en la expresión regular que estamos trabajando, con un +, notese que ese “mas” esta fungiendo como OR, nos da la posibilidad de usar uno u otro, a si que este no esta “sumando” a los números, una vez hecha esa aclaración, tendremos que:

$$((-|\epsilon)digit^+) + ((-|\epsilon)digit^+.digit^+)$$

Ahora, analizando la parte de :

$$(e(+| -|\epsilon)digitdigit^*|\epsilon)$$

Tenemos un problema en la parte de $(+| -|\epsilon)$, por las razones revisadas anteriormente, ya que recordemos que si tenemos por ejemplo al $1.52e + 12$, con eso nos estamos refiriendo a 1.52×10^{12} (notemos como esto puede ser logrado igualmente con $1.52e12$), teniendo de esta forma a $(\epsilon| -)$ como nuestra solución.

Con todo lo anterior podemos concluir que una expresión regular mas próxima a nuestro objetivo de reconocer los numeros reales es la de:

$$(((-|\epsilon)digit^+) + ((-|\epsilon)digit^+.digit^+))(e(-|\epsilon)digit^+|\epsilon)$$



3. Considera el lenguaje Pascal

- a) [1.5pts] Las siguientes expresiones regulares definen algunos tokens. Utilízalas para definir una expresión para representar números reales.

```
digit      [0-9]
digits     {digit}+
sign       [+|-]
dtdgts    {dot}{digits}
exponent  {Ee}{sign}?{digits}
dot        ``.''
```

Para este ejercicio desarrollaremos la expresión regular por pasos, por lo que la obtendremos de la siguiente forma:

Primero, notemos lo siguiente:

- digit: Define un solo dígito del 0 al 9.
- digits: Representa una secuencia de uno o más dígitos.
- sign: Define un signo positivo o negativo.
- exponent: Define la notación exponencial en la forma E o e seguida opcionalmente por un signo y uno o más dígitos.
- dot: Representa un punto decimal.

Tomando eso en cuenta, primero debemos crear la parte que maneja la parte entera y decimal del número real. Puede haber un signo seguido de uno o más dígitos, y también puede haber un punto seguido de uno o más dígitos para representar la parte fraccionaria, esto es:

$$(([+|-]\{[0-9]\}+)|([+|-]\{[0-9]\} + \{.\}\{[0-9]\}+))$$

Ahora, también debemos definir la parte que maneja la parte exponencial del número real. Puede haber una E o e seguida opcionalmente por un signo y uno o más dígitos, esto es:

$$(\varepsilon|{Ee}\{[+|-]\}?\{[0-9]\}+)$$

Por lo tanto, la expresión final para representar números reales que define a los tokens dados es:

$$(([+|-]\{[0-9]\}+)|([+|-]\{[0-9]\} + \{.\}\{[0-9]\}+))(\varepsilon|{Ee}\{[+|-]\}?\{[0-9]\}+)$$

- b) [2.5pts] Un comentario está delimitado por { y } o también por (* y *). Los comentarios no pueden estar anidados.

Construye un autómata finito, de preferencia determinista, que reconoce los comentarios de Pascal y obtén la expresión regular (puedes usar algún método, eg. derivadas de expresiones regulares o construcción de un AFN y transformaciones pero debes indicar el método usado y mostrar el proceso).

Usaremos el método de derivaciones en expresiones regulares, de esta manera daremos la expresión regular primero y luego construimos el autómata.

Primero comenzemos con una expresión para reconocer un comentario que empieza con (*) y termina con (*), sin tener un comentario de la misma forma anidado. Para eso tenemos esta expresión:

$$\backslash(*\left(\left[\wedge*\right]^*\ast^+\left[\wedge*\right]\right)\left.\right)^*\left[\wedge*\right]^*\ast^+\backslash)$$

Lo cual sería algo como un (seguido de *, luego 0 o más veces esto 0 o más veces cualquier carácter menos * (el significado de [^] sería como complemento, como en la presentación 2 de la clase) seguido de 1 o más veces * y terminado con cualquier carácter menos la combinación *), después 0 o más veces cualquier carácter menos *, luego 1 o más veces * y por último)

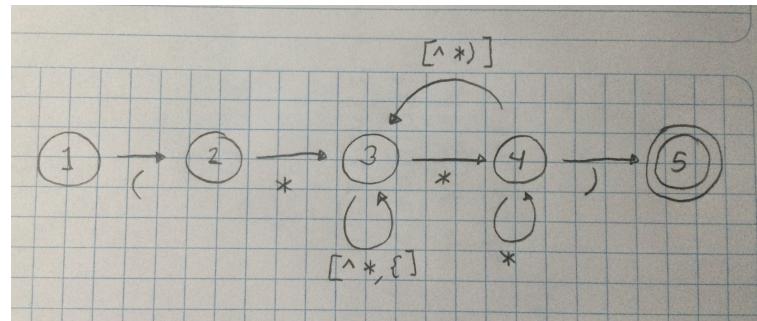


De esta manera podemos reconocer comentarios que empiecen con (*) y terminen con (*), sin tener un comentario de la misma forma anidado, pero nos falta revisar que no tenga un comentario que empiece con { y termine } anidado, para esto tenemos dos maneras, la primera es revisar que dentro del comentario no tengamos ningún { (es decir, que nunca se abra un comentario de ese tipo) o que no tengamos ningún } (es decir, que nunca se cierre un comentario de ese tipo), de esta forma obtenemos dos expresiones regulares, en una versión agregamos { a dentro de la expresión [^], para que sea cualquier carácter menos { y los otros dentro de la expresión, y la otra versión es similar solo con }, de esta forma nos quedan estas dos expresiones y sus autómatas respectivos.

★ Expresión 1:

$$\backslash(* ([^*, \{]^* *+ [^*\backslash], \{])^* [^*, \{]^* *+ \backslash)$$

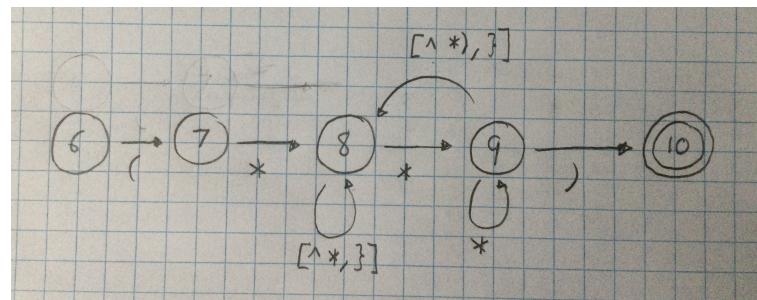
➤ Autómata Expresión 1:



★ Expresión 2:

$$\backslash(* ([^*, \}]^* *+ [^*\backslash], \}])^* [^*, \}]^* *+ \backslash)$$

➤ Autómata Expresión 2:



Con estas dos expresiones entonces podemos reconocer un comentario que empieza con (*) y termina con (*), sin tener comentarios anidados de ningún tipo, entonces ahora nos falta como reconocer los comentarios { }

Empecemos notando como reconocer un comentario que empieza con { y termine con }, sin tener un comentario de este mismo tipo anidado, la expresión es esta:

$$\{ [^ \}]^* \}$$

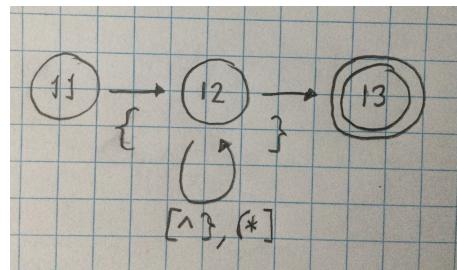


Lo cual seria algo como un { seguido 0 o más veces cualquier carácter menos } y por ultimo un } es una expresión sencilla, ahora nos falta evitar comentarios (* *) anidados, para esto lo vamos a hacer igual que hicimos en los otros comentarios, es decir hacer dos expresiones una sin (*) y otra sin *), entonces las expresión quedan así, con sus respectivos autómatas

★ Expresión 3:

$$\{ [^*], \backslash(*)^* \}$$

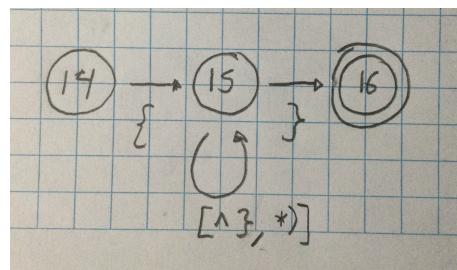
➤ Autómata Expresión 3:



★ Expresión 4:

$$\{ [^*], *\backslash)]^* \}$$

➤ Autómata Expresión 4:

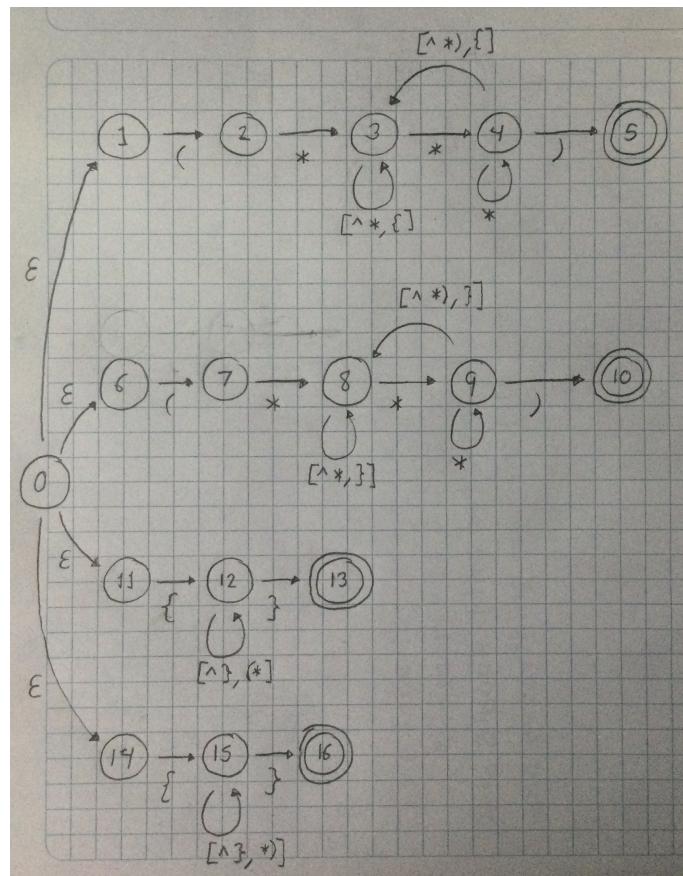


Por lo tanto tenemos 4 expresiones y autómatas para reconocer comentarios, por ultimo solo los unimos todos, las expresiones las unimos con | y los autómatas con transiciones ϵ , terminando en

★ Expresión:

$$\backslash(* ([^*, \{]^* *^+ [^*\backslash), \{])^* [^*, \{]^* *^+ \backslash) | \backslash(* ([^*, \{]^* *^+ [^*\backslash), \{])^* [^*, \{]^* *^+ \backslash) | \\{ [^*], \backslash(*)^* \} | \{ [^*], *\backslash]^* \}$$

➤ Autómata:



Y así quedo para reconocer los comentarios (o al menos espero que así sea), nos quedo un autómata no determinista, se podría hacer una expresión y autómata mas elegantes pero así quedo.



4. [2.5pts] Considera la siguiente definición para un analizador léxico donde se especifican los tokens válidos y las acciones a realizar cuando se identifique alguno de ellos:

```
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" }
```

donde **lexbuf** es un *buffer* para almacenar el estado actual del scanner el cual mantiene la posición de lectura en la cadena de entrada, en particular **lexeme lexbuf** devuelve una cadena que coincide con una expresión regular.

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

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

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

fun c ->e c	Lexema	Token	Valor	# tokens
-------------	--------	-------	-------	----------

Procedemos a analizar las cadenas:

fun x ->z x	Lexema	Token	Valor	
	'fun'	FUN		
	'x'	VAR	x	
	'->'	ARR		
	'z'	VAR	z	
	'x'	VAR	x	5 tokens

(fun w ->w w)	Lexema	Token	Valor	
	'('	LPAREN		
	'fun'	FUN		
	'w'	VAR	w	
	'->'	ARR		
	'w'	VAR	w	
	'w'	VAR	w	
	')'	RPAREN		7 tokens



Referencias

- [1] What is a programming language?
- [2] What is high level language?
- [3] What is low level language?
- [4] Recognising strings and/or comments