LENGUAJES FORMALES

|  |  |
| --- | --- |
| *Integrantes:* R*odríguez Castro Carlos Eduardo, Cordero Hernández Marco Ricardo* | **Fecha 07/12/2021** |

Proyecto final

Tabla de contenido

[Parte I. Investigación. 1](#_Toc89157465)

[Parte II. Sintaxis lenguajes de alto nivel 1](#_Toc89157466)

[Parte III. Caso de estudio 1](#_Toc89157467)

[Especificaciones 2](#_Toc89157468)

[Entrada: 2](#_Toc89157469)

[Salida: 2](#_Toc89157470)

[Reporte de resultados 3](#_Toc89157471)

[Paso 1. 3](#_Toc89157472)

[Paso 2. 3](#_Toc89157473)

[Paso 3. 3](#_Toc89157474)

[Paso 4. 3](#_Toc89157475)

[Conclusiones. 4](#_Toc89157476)

[Bibliografía. 4](#_Toc89157477)

## Parte I. Investigación aplicaciones.

La primera aplicación de los autómatas programables fue reemplazar equipos complejos basados ​​en relés en la industria automotriz, pero debido a las reducciones de tamaño y al ahorro de costos ahora los autómatas se están extendiendo a todos los sectores de las industrias que utilizan software como medio de control de flujo.

Algunos ejemplos del uso de autómatas en la industria de la fabricación de neumáticos son:

* Control de calderas
* Sistemas de refrigeración
* Control de máquinas para mezclar caucho

Estos solo son una pequeña cantidad de ejemplos, ya que la mayoría de los sistemas utilizan de manera directa o indirecta los autómatas.

Las expresiones regulares son utilizadas en gran parte por los programadores para hacer validación de entradas y búsqueda específica de información en grandes sets de datos para el análisis de estos. Estas son muy importantes a la hora de validar entradas que se les dan a los programas, ya sea validar un email al ingresarlo para iniciar sesión en tu cuenta hasta la recopilación de repeticiones de datos dentro de un dataset específico.

## Parte II. Sintaxis lenguajes de alto nivel

* Python

# PEG grammar for Python

file: [statements] ENDMARKER

interactive: statement\_newline

eval: expressions NEWLINE\* ENDMARKER

func\_type: '(' [type\_expressions] ')' '->' expression NEWLINE\* ENDMARKER

fstring: star\_expressions

# type\_expressions allow \*/\*\* but ignore them

type\_expressions:

| ','.expression+ ',' '\*' expression ',' '\*\*' expression

| ','.expression+ ',' '\*' expression

| ','.expression+ ',' '\*\*' expression

| '\*' expression ',' '\*\*' expression

| '\*' expression

| '\*\*' expression

| ','.expression+

statements: statement+

statement: compound\_stmt | simple\_stmts

statement\_newline:

| compound\_stmt NEWLINE

| simple\_stmts

| NEWLINE

| ENDMARKER

simple\_stmts:

| simple\_stmt !';' NEWLINE # Not needed, there for speedup

| ';'.simple\_stmt+ [';'] NEWLINE

# NOTE: assignment MUST precede expression, else parsing a simple assignment

# will throw a SyntaxError.

simple\_stmt:

| assignment

| star\_expressions

| return\_stmt

| import\_stmt

| raise\_stmt

| 'pass'

| del\_stmt

| yield\_stmt

| assert\_stmt

| 'break'

| 'continue'

| global\_stmt

| nonlocal\_stmt

compound\_stmt:

| function\_def

| if\_stmt

| class\_def

| with\_stmt

| for\_stmt

| try\_stmt

| while\_stmt

| match\_stmt

# NOTE: annotated\_rhs may start with 'yield'; yield\_expr must start with 'yield'

assignment:

| NAME ':' expression ['=' annotated\_rhs ]

| ('(' single\_target ')'

| single\_subscript\_attribute\_target) ':' expression ['=' annotated\_rhs ]

| (star\_targets '=' )+ (yield\_expr | star\_expressions) !'=' [TYPE\_COMMENT]

| single\_target augassign ~ (yield\_expr | star\_expressions)

augassign:

| '+='

| '-='

| '\*='

| '@='

| '/='

| '%='

| '&='

| '|='

| '^='

| '<<='

| '>>='

| '\*\*='

| '//='

global\_stmt: 'global' ','.NAME+

nonlocal\_stmt: 'nonlocal' ','.NAME+

yield\_stmt: yield\_expr

assert\_stmt: 'assert' expression [',' expression ]

del\_stmt:

| 'del' del\_targets &(';' | NEWLINE)

import\_stmt: import\_name | import\_from

import\_name: 'import' dotted\_as\_names

# note below: the ('.' | '...') is necessary because '...' is tokenized as ELLIPSIS

import\_from:

| 'from' ('.' | '...')\* dotted\_name 'import' import\_from\_targets

| 'from' ('.' | '...')+ 'import' import\_from\_targets

import\_from\_targets:

| '(' import\_from\_as\_names [','] ')'

| import\_from\_as\_names !','

| '\*'

import\_from\_as\_names:

| ','.import\_from\_as\_name+

import\_from\_as\_name:

| NAME ['as' NAME ]

dotted\_as\_names:

| ','.dotted\_as\_name+

dotted\_as\_name:

| dotted\_name ['as' NAME ]

dotted\_name:

| dotted\_name '.' NAME

| NAME

if\_stmt:

| 'if' named\_expression ':' block elif\_stmt

| 'if' named\_expression ':' block [else\_block]

elif\_stmt:

| 'elif' named\_expression ':' block elif\_stmt

| 'elif' named\_expression ':' block [else\_block]

else\_block:

| 'else' ':' block

while\_stmt:

| 'while' named\_expression ':' block [else\_block]

for\_stmt:

| 'for' star\_targets 'in' ~ star\_expressions ':' [TYPE\_COMMENT] block [else\_block]

| ASYNC 'for' star\_targets 'in' ~ star\_expressions ':' [TYPE\_COMMENT] block [else\_block]

with\_stmt:

| 'with' '(' ','.with\_item+ ','? ')' ':' block

| 'with' ','.with\_item+ ':' [TYPE\_COMMENT] block

| ASYNC 'with' '(' ','.with\_item+ ','? ')' ':' block

| ASYNC 'with' ','.with\_item+ ':' [TYPE\_COMMENT] block

with\_item:

| expression 'as' star\_target &(',' | ')' | ':')

| expression

try\_stmt:

| 'try' ':' block finally\_block

| 'try' ':' block except\_block+ [else\_block] [finally\_block]

except\_block:

| 'except' expression ['as' NAME ] ':' block

| 'except' ':' block

finally\_block:

| 'finally' ':' block

match\_stmt:

| "match" subject\_expr ':' NEWLINE INDENT case\_block+ DEDENT

subject\_expr:

| star\_named\_expression ',' star\_named\_expressions?

| named\_expression

case\_block:

| "case" patterns guard? ':' block

guard: 'if' named\_expression

patterns:

| open\_sequence\_pattern

| pattern

pattern:

| as\_pattern

| or\_pattern

as\_pattern:

| or\_pattern 'as' pattern\_capture\_target

or\_pattern:

| '|'.closed\_pattern+

closed\_pattern:

| literal\_pattern

| capture\_pattern

| wildcard\_pattern

| value\_pattern

| group\_pattern

| sequence\_pattern

| mapping\_pattern

| class\_pattern

# Literal patterns are used for equality and identity constraints

literal\_pattern:

| signed\_number !('+' | '-')

| complex\_number

| strings

| 'None'

| 'True'

| 'False'

# Literal expressions are used to restrict permitted mapping pattern keys

literal\_expr:

| signed\_number !('+' | '-')

| complex\_number

| strings

| 'None'

| 'True'

| 'False'

complex\_number:

| signed\_real\_number '+' imaginary\_number

| signed\_real\_number '-' imaginary\_number

signed\_number:

| NUMBER

| '-' NUMBER

signed\_real\_number:

| real\_number

| '-' real\_number

real\_number:

| NUMBER

imaginary\_number:

| NUMBER

capture\_pattern:

| pattern\_capture\_target

pattern\_capture\_target:

| !"\_" NAME !('.' | '(' | '=')

wildcard\_pattern:

| "\_"

value\_pattern:

| attr !('.' | '(' | '=')

attr:

| name\_or\_attr '.' NAME

name\_or\_attr:

| attr

| NAME

group\_pattern:

| '(' pattern ')'

sequence\_pattern:

| '[' maybe\_sequence\_pattern? ']'

| '(' open\_sequence\_pattern? ')'

open\_sequence\_pattern:

| maybe\_star\_pattern ',' maybe\_sequence\_pattern?

maybe\_sequence\_pattern:

| ','.maybe\_star\_pattern+ ','?

maybe\_star\_pattern:

| star\_pattern

| pattern

star\_pattern:

| '\*' pattern\_capture\_target

| '\*' wildcard\_pattern

mapping\_pattern:

| '{' '}'

| '{' double\_star\_pattern ','? '}'

| '{' items\_pattern ',' double\_star\_pattern ','? '}'

| '{' items\_pattern ','? '}'

items\_pattern:

| ','.key\_value\_pattern+

key\_value\_pattern:

| (literal\_expr | attr) ':' pattern

double\_star\_pattern:

| '\*\*' pattern\_capture\_target

class\_pattern:

| name\_or\_attr '(' ')'

| name\_or\_attr '(' positional\_patterns ','? ')'

| name\_or\_attr '(' keyword\_patterns ','? ')'

| name\_or\_attr '(' positional\_patterns ',' keyword\_patterns ','? ')'

positional\_patterns:

| ','.pattern+

keyword\_patterns:

| ','.keyword\_pattern+

keyword\_pattern:

| NAME '=' pattern

return\_stmt:

| 'return' [star\_expressions]

raise\_stmt:

| 'raise' expression ['from' expression ]

| 'raise'

function\_def:

| decorators function\_def\_raw

| function\_def\_raw

function\_def\_raw:

| 'def' NAME '(' [params] ')' ['->' expression ] ':' [func\_type\_comment] block

| ASYNC 'def' NAME '(' [params] ')' ['->' expression ] ':' [func\_type\_comment] block

func\_type\_comment:

| NEWLINE TYPE\_COMMENT &(NEWLINE INDENT) # Must be followed by indented block

| TYPE\_COMMENT

params:

| parameters

parameters:

| slash\_no\_default param\_no\_default\* param\_with\_default\* [star\_etc]

| slash\_with\_default param\_with\_default\* [star\_etc]

| param\_no\_default+ param\_with\_default\* [star\_etc]

| param\_with\_default+ [star\_etc]

| star\_etc

# Some duplication here because we can't write (',' | &')'),

# which is because we don't support empty alternatives (yet).

#

slash\_no\_default:

| param\_no\_default+ '/' ','

| param\_no\_default+ '/' &')'

slash\_with\_default:

| param\_no\_default\* param\_with\_default+ '/' ','

| param\_no\_default\* param\_with\_default+ '/' &')'

star\_etc:

| '\*' param\_no\_default param\_maybe\_default\* [kwds]

| '\*' ',' param\_maybe\_default+ [kwds]

| kwds

kwds: '\*\*' param\_no\_default

# One parameter. This \*includes\* a following comma and type comment.

#

# There are three styles:

# - No default

# - With default

# - Maybe with default

#

# There are two alternative forms of each, to deal with type comments:

# - Ends in a comma followed by an optional type comment

# - No comma, optional type comment, must be followed by close paren

# The latter form is for a final parameter without trailing comma.

#

param\_no\_default:

| param ',' TYPE\_COMMENT?

| param TYPE\_COMMENT? &')'

param\_with\_default:

| param default ',' TYPE\_COMMENT?

| param default TYPE\_COMMENT? &')'

param\_maybe\_default:

| param default? ',' TYPE\_COMMENT?

| param default? TYPE\_COMMENT? &')'

param: NAME annotation?

annotation: ':' expression

default: '=' expression

decorators: ('@' named\_expression NEWLINE )+

class\_def:

| decorators class\_def\_raw

| class\_def\_raw

class\_def\_raw:

| 'class' NAME ['(' [arguments] ')' ] ':' block

block:

| NEWLINE INDENT statements DEDENT

| simple\_stmts

star\_expressions:

| star\_expression (',' star\_expression )+ [',']

| star\_expression ','

| star\_expression

star\_expression:

| '\*' bitwise\_or

| expression

star\_named\_expressions: ','.star\_named\_expression+ [',']

star\_named\_expression:

| '\*' bitwise\_or

| named\_expression

assignment\_expression:

| NAME ':=' ~ expression

named\_expression:

| assignment\_expression

| expression !':='

annotated\_rhs: yield\_expr | star\_expressions

expressions:

| expression (',' expression )+ [',']

| expression ','

| expression

expression:

| disjunction 'if' disjunction 'else' expression

| disjunction

| lambdef

lambdef:

| 'lambda' [lambda\_params] ':' expression

lambda\_params:

| lambda\_parameters

# lambda\_parameters etc. duplicates parameters but without annotations

# or type comments, and if there's no comma after a parameter, we expect

# a colon, not a close parenthesis. (For more, see parameters above.)

#

lambda\_parameters:

| lambda\_slash\_no\_default lambda\_param\_no\_default\* lambda\_param\_with\_default\* [lambda\_star\_etc]

| lambda\_slash\_with\_default lambda\_param\_with\_default\* [lambda\_star\_etc]

| lambda\_param\_no\_default+ lambda\_param\_with\_default\* [lambda\_star\_etc]

| lambda\_param\_with\_default+ [lambda\_star\_etc]

| lambda\_star\_etc

lambda\_slash\_no\_default:

| lambda\_param\_no\_default+ '/' ','

| lambda\_param\_no\_default+ '/' &':'

lambda\_slash\_with\_default:

| lambda\_param\_no\_default\* lambda\_param\_with\_default+ '/' ','

| lambda\_param\_no\_default\* lambda\_param\_with\_default+ '/' &':'

lambda\_star\_etc:

| '\*' lambda\_param\_no\_default lambda\_param\_maybe\_default\* [lambda\_kwds]

| '\*' ',' lambda\_param\_maybe\_default+ [lambda\_kwds]

| lambda\_kwds

lambda\_kwds: '\*\*' lambda\_param\_no\_default

lambda\_param\_no\_default:

| lambda\_param ','

| lambda\_param &':'

lambda\_param\_with\_default:

| lambda\_param default ','

| lambda\_param default &':'

lambda\_param\_maybe\_default:

| lambda\_param default? ','

| lambda\_param default? &':'

lambda\_param: NAME

disjunction:

| conjunction ('or' conjunction )+

| conjunction

conjunction:

| inversion ('and' inversion )+

| inversion

inversion:

| 'not' inversion

| comparison

comparison:

| bitwise\_or compare\_op\_bitwise\_or\_pair+

| bitwise\_or

compare\_op\_bitwise\_or\_pair:

| eq\_bitwise\_or

| noteq\_bitwise\_or

| lte\_bitwise\_or

| lt\_bitwise\_or

| gte\_bitwise\_or

| gt\_bitwise\_or

| notin\_bitwise\_or

| in\_bitwise\_or

| isnot\_bitwise\_or

| is\_bitwise\_or

eq\_bitwise\_or: '==' bitwise\_or

noteq\_bitwise\_or:

| ('!=' ) bitwise\_or

lte\_bitwise\_or: '<=' bitwise\_or

lt\_bitwise\_or: '<' bitwise\_or

gte\_bitwise\_or: '>=' bitwise\_or

gt\_bitwise\_or: '>' bitwise\_or

notin\_bitwise\_or: 'not' 'in' bitwise\_or

in\_bitwise\_or: 'in' bitwise\_or

isnot\_bitwise\_or: 'is' 'not' bitwise\_or

is\_bitwise\_or: 'is' bitwise\_or

bitwise\_or:

| bitwise\_or '|' bitwise\_xor

| bitwise\_xor

bitwise\_xor:

| bitwise\_xor '^' bitwise\_and

| bitwise\_and

bitwise\_and:

| bitwise\_and '&' shift\_expr

| shift\_expr

shift\_expr:

| shift\_expr '<<' sum

| shift\_expr '>>' sum

| sum

sum:

| sum '+' term

| sum '-' term

| term

term:

| term '\*' factor

| term '/' factor

| term '//' factor

| term '%' factor

| term '@' factor

| factor

factor:

| '+' factor

| '-' factor

| '~' factor

| power

power:

| await\_primary '\*\*' factor

| await\_primary

await\_primary:

| AWAIT primary

| primary

primary:

| primary '.' NAME

| primary genexp

| primary '(' [arguments] ')'

| primary '[' slices ']'

| atom

slices:

| slice !','

| ','.slice+ [',']

slice:

| [expression] ':' [expression] [':' [expression] ]

| named\_expression

atom:

| NAME

| 'True'

| 'False'

| 'None'

| strings

| NUMBER

| (tuple | group | genexp)

| (list | listcomp)

| (dict | set | dictcomp | setcomp)

| '...'

strings: STRING+

list:

| '[' [star\_named\_expressions] ']'

listcomp:

| '[' named\_expression for\_if\_clauses ']'

tuple:

| '(' [star\_named\_expression ',' [star\_named\_expressions] ] ')'

group:

| '(' (yield\_expr | named\_expression) ')'

genexp:

| '(' ( assignment\_expression | expression !':=') for\_if\_clauses ')'

set: '{' star\_named\_expressions '}'

setcomp:

| '{' named\_expression for\_if\_clauses '}'

dict:

| '{' [double\_starred\_kvpairs] '}'

| '{' invalid\_double\_starred\_kvpairs '}'

dictcomp:

| '{' kvpair for\_if\_clauses '}'

double\_starred\_kvpairs: ','.double\_starred\_kvpair+ [',']

double\_starred\_kvpair:

| '\*\*' bitwise\_or

| kvpair

kvpair: expression ':' expression

for\_if\_clauses:

| for\_if\_clause+

for\_if\_clause:

| ASYNC 'for' star\_targets 'in' ~ disjunction ('if' disjunction )\*

| 'for' star\_targets 'in' ~ disjunction ('if' disjunction )\*

yield\_expr:

| 'yield' 'from' expression

| 'yield' [star\_expressions]

arguments:

| args [','] &')'

args:

| ','.(starred\_expression | ( assignment\_expression | expression !':=') !'=')+ [',' kwargs ]

| kwargs

kwargs:

| ','.kwarg\_or\_starred+ ',' ','.kwarg\_or\_double\_starred+

| ','.kwarg\_or\_starred+

| ','.kwarg\_or\_double\_starred+

starred\_expression:

| '\*' expression

kwarg\_or\_starred:

| NAME '=' expression

| starred\_expression

kwarg\_or\_double\_starred:

| NAME '=' expression

| '\*\*' expression

# NOTE: star\_targets may contain \*bitwise\_or, targets may not.

star\_targets:

| star\_target !','

| star\_target (',' star\_target )\* [',']

star\_targets\_list\_seq: ','.star\_target+ [',']

star\_targets\_tuple\_seq:

| star\_target (',' star\_target )+ [',']

| star\_target ','

star\_target:

| '\*' (!'\*' star\_target)

| target\_with\_star\_atom

target\_with\_star\_atom:

| t\_primary '.' NAME !t\_lookahead

| t\_primary '[' slices ']' !t\_lookahead

| star\_atom

star\_atom:

| NAME

| '(' target\_with\_star\_atom ')'

| '(' [star\_targets\_tuple\_seq] ')'

| '[' [star\_targets\_list\_seq] ']'

single\_target:

| single\_subscript\_attribute\_target

| NAME

| '(' single\_target ')'

single\_subscript\_attribute\_target:

| t\_primary '.' NAME !t\_lookahead

| t\_primary '[' slices ']' !t\_lookahead

del\_targets: ','.del\_target+ [',']

del\_target:

| t\_primary '.' NAME !t\_lookahead

| t\_primary '[' slices ']' !t\_lookahead

| del\_t\_atom

del\_t\_atom:

| NAME

| '(' del\_target ')'

| '(' [del\_targets] ')'

| '[' [del\_targets] ']'

t\_primary:

| t\_primary '.' NAME &t\_lookahead

| t\_primary '[' slices ']' &t\_lookahead

| t\_primary genexp &t\_lookahead

| t\_primary '(' [arguments] ')' &t\_lookahead

| atom &t\_lookahead

t\_lookahead: '(' | '[' | '.'

- SQL

<Query> ::= SELECT <SelList>

FROM <FromList>

WHERE <Condition>

<SelList> ::= <Attribute> |

<Attribute> , <SelList>

<FromList> ::= <Relation> |

<Relation> , <FromList>

<Condition> ::= <Condition> AND <Condition> |

<Attribute> IN ( <Query> ) |

<Attribute> = <Attribute> |

<Attribute> LIKE <Pattern>

- LISP

<grail-list> ::= "'(" {<grail-rule>} ")"

<grail-rule> ::= <assignment> | <alternation>

<assignment> ::= "(" <type> " ::= " <s-exp> ")"

<alternation> ::= "(" <type> " ::= " <type> {<type>} ")"

<s-exp> ::= <symbol> | <nonterminal> | "(" {<s-exp>} ")"

<type> ::= "#(" <type-name> ")"

<nonterminal> ::= "#(" {<arg-name> " "} <type-name> ")"

<type-name> ::= <symbol>

<arg-name> ::= <symbol>

## Parte III. Caso de estudio

Objetivo

Resolver un problema que requiera el reconocimiento de patrones de texto y estructuras gramaticales, utilizando software de generación de analizadores léxicos y sintácticos.

Enunciado del problema

Crear un programa, utilizando el generador de analizadores léxicos JFlex y el de analizadores sintácticos CUP, que transforme una expresión regular a notación postfija.

## Especificaciones

La sintaxis para definir las expresiones regulares será de la siguiente manera:

Una ER válida será válida de la siguiente forma:

Base:

La cadena vacía ε, representada como $

Cualquier símbolo del alfabeto mayúscula o minúscula y cualquier dígito es una ER.

Inductivo

1. Si E y F son expresiones regulares, entonces E,F también lo es y representa la unión de L(E)∪L(F).
2. Si E y F son expresiones regulares, entonces EF también lo es y representa la concatenación de L(E)L(F). (Para imprimirlo como operador en la expresión postfija utilizaremos el punto)
3. Si E es una expresión regular, entonces E∗ también lo es y representa L(E)∗.
4. Si E es una expresión regular, entonces (E), también lo es.

### Entrada:

Serán un listado de expresiones regulares separados por un punto y coma.

a\*;

a , b c ;

a,(bca,a)\*;

a,(bca,a)\*\*;

a,(bca,a)\*,K\*;

### Salida:

Transformación a notación postfija.

a \*

a b c . ,

a b c . a . a , \* ,

a b c . a . a , \* \* ,

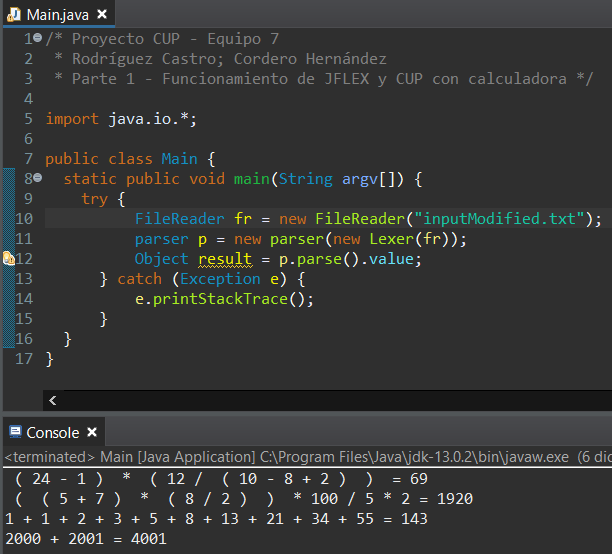
a b c . a . a , \* , K \* ,

## Reporte de resultados

### Paso 1.

Poner en funcionamiento las herramientas de JFLEX y CUP con el ejemplo de una calculadora de números enteros.

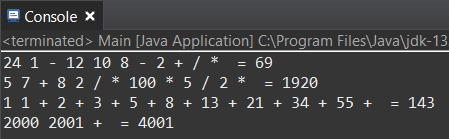
a. Insertar captura de pantalla con una entrada distinta a la asignada en el ejemplo de los archivos descargados.



### Paso 2.

Modificar la calculadora de números enteros para generar en notación postfija cada una de las expresiones aritméticas en el archivo input.txt

b. Inserta una captura de pantalla con la misma entrada en el inciso a.



### 

### Paso 3.

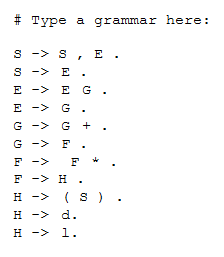
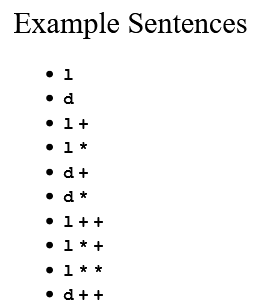
Genera la gramática para generar expresiones regulares, recuerda que se comporta de manera muy similar a la de aritmética, sólo hay que considerar los nuevos operadores y su jerarquía.

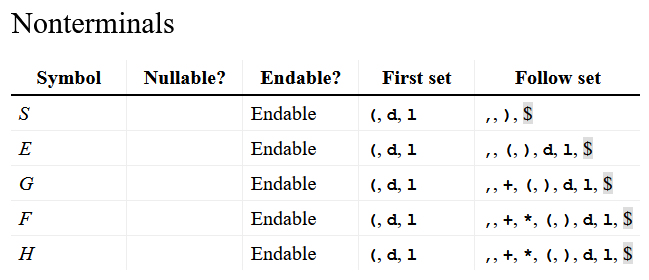
c. Inserta la gramática nueva con formato solicitado en la página, adicional, muestra capturas de pantalla en este inciso con los resultados que muestra la página sobre tu gramática: <https://mdaines.github.io/grammophone/>

Donde

d = Dígito

l = Letra







### Paso 4.

Modifica los archivos input.txt, ycalc.cup, lcalc.flex para que ahora generen la notación postfija de la expresión regular.

d. Lcalc.flex modificado

import java\_cup.runtime.\*;

%%

%class Lexer

%line

%column

%cup

%{

private Symbol symbol(int type) {

return new Symbol(type, yyline, yycolumn);

}

private Symbol symbol(int type, Object value) {

return new Symbol(type, yyline, yycolumn, value);

}

%}

LineTerminator = \r|\n|\r\n

WhiteSpace = {LineTerminator} | [ \t\f]

letra = [A-Za-z]

digito= [0-9]

%%

/\* YYINITIAL is the state \*/

<YYINITIAL> {

";" { return symbol(sym.SEMI); }

"," { return symbol(sym.UNION); }

"\*" { return symbol(sym.KLEENE); }

"(" { return symbol(sym.LPAREN); }

")" { return symbol(sym.RPAREN); }

"+" { return symbol(sym.POS); }

"$" { return symbol(sym.EPSILON); }

{letra} { return symbol(sym.SYMB, new String(yytext())); }

{digito} { return symbol(sym.DIGIT, new String(yytext())); }

{WhiteSpace} { /\* do nothing \*/ }

}

[^] { throw new Error("Illegal character <"+yytext()+">"); }

e. Ycalc.cup modificado

import java\_cup.runtime.\*;

action code {:

public String x = "" ;

:}

parser code {:

public void report\_error(String message, Object info) {

StringBuilder m = new StringBuilder("Error");

if (info instanceof java\_cup.runtime.Symbol) {

java\_cup.runtime.Symbol s = ((java\_cup.runtime.Symbol) info);

if (s.left >= 0) {

m.append(" in line "+(s.left+1));

if (s.right >= 0)

m.append(", column "+(s.right+1));

}

}

m.append(" : "+message);

System.err.println(m);

}

public void report\_fatal\_error(String message, Object info) {

report\_error(message, info);

System.exit(1);

}

:};

/\* ------------Declaration of Terminals and Non Terminals Section----------- \*/

terminal SEMI, UNION, KLEENE, LPAREN, RPAREN, POS, EPSILON, SYMB, DIGIT;

non terminal Object expr\_list, expr\_part, S, E, G, F, H;

/\* -------------Precedence and Associatively of Terminals Section----------- \*/

precedence left KLEENE;

precedence left UNION;

/\* ----------------------------Grammar Section-------------------- \*/

/\*----------------------------------------------

expr\_list ::= expr\_list expr\_part

| expr\_part

expr\_part ::= S SEMI

S ::= S UNION E

| E

E ::= E G

| G

G ::= G POS

| F

F ::= F KLEENE

| H

H ::= LPAREN S RPAREN

| DIGIT

| SYMB

| EPSILON

----------------------------------------------------\*/

expr\_list ::= expr\_list expr\_part

|

expr\_part

;

expr\_part ::= S:s

{: System.out.println(x); x = ""; :}

SEMI

;

S ::= S:s UNION E:e

{: x += ", "; :}

|

E:e

{: RESULT = e; :}

;

E ::= E:e G:g

{: x += ". "; :}

|

G:g

{: RESULT = g; :}

;

G ::= G:g POS

{: x += "+ "; :}

|

F:f

{: RESULT = f; :}

;

F ::= F:f KLEENE

{: x += "\* "; :}

|

H:h

{: RESULT = h; :}

;

H ::= LPAREN S:s RPAREN

{: RESULT = s; :}

|

DIGIT:d

{: x += d + " "; RESULT = d; :}

|

SYMB:s

{: x += s + " "; RESULT = s; :}

|

EPSILON:e

{: x += "$ "; RESULT = e; :}

;

f. Input.txt (ejemplos necesarios, anota más ejemplos conforme a tus pruebas)

a\*;

a , b c ;

a,(bca,a)\*;

a,(bca,a)\*\*;

a,(bca,a)\*,K\*;

$;

8\*,A;

((A24,ZZZ)\*\*)+ITUP+AT;

4+5,19;

Un conjunto de letras negras en un fondo negro

Descripción generada automáticamente con confianza mediag. Captura de pantalla que genere la notación postfija de las expresiones regulares en input.txt

## Conclusiones.

Los resultados que obtuvimos a través de aplicar los métodos aprendidos en clase fueron precisos y objetivos, mostrando la respuesta correcta después de varios intentos no exitosos. La clave de nuestro éxito fue hacer pruebas con distintas maneras de resolver el problema hasta encontrar una lo suficientemente madura y refinada para ser implementada. El aprendizaje obtenido de este proyecto nos servirá en un futuro a saber analizar con mayor profundidad el comportamiento de los lenguajes de programación y el cómo se construyen estos. Una recomendación que como equipo queremos hacer es ser más específicos a la hora de enseñar la herramienta jflex y cup, ya que no aprendimos la sintaxis como tal de las herramientas, sino solo a ver los parsers ya hechos por la maestra y a partir de esto se editaba y trabajaba, algo que no sucede en un ambiente laboral profesional.

## Bibliografía.

Diego Zerkk. (2020). Aplicaciones Autómatas. 06-12-2021, de GoConqr Sitio web: <https://www.goconqr.com/es/mindmap/3684210/aplicaciones-automatas>.

Python.org. (s.f.). *Full Grammar Specification*. Recuperado de <https://docs.python.org/3/reference/grammar.html>.

*A simplified SQL grammar*. (s.f.). Recuperado de <http://www.mathcs.emory.edu/~cheung/Courses/554/Syllabus/5-query-opt/SQL-grammar.html>.

Bertram. G. (2017). *GRAIL (GRAmmar In Lisp)*. Recuperado de <https://stackoverflow.com/questions/36189547/common-lisp-a-good-way-to-represent-grammar-rules>.