

LENGUAJES FORMALES

*Integrantes: Rodríguez Castro Carlos Eduardo, Cordero
Hernández Marco Ricardo*

Fecha 07/12/2021

Proyecto final

Tabla de contenido

Parte I. Investigación.....	2
Parte II. Sintaxis lenguajes de alto nivel	3
Parte III. Caso de estudio	11
Especificaciones	11
Entrada:.....	11
Salida:	11
Reporte de resultados	12
Paso 1.	12
Paso 2.	12
Paso 3.	13
Paso 4.	14
Conclusiones.	18
Bibliografía.	19

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 ]
    | '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 $L(E) \cup L(F)$.
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.

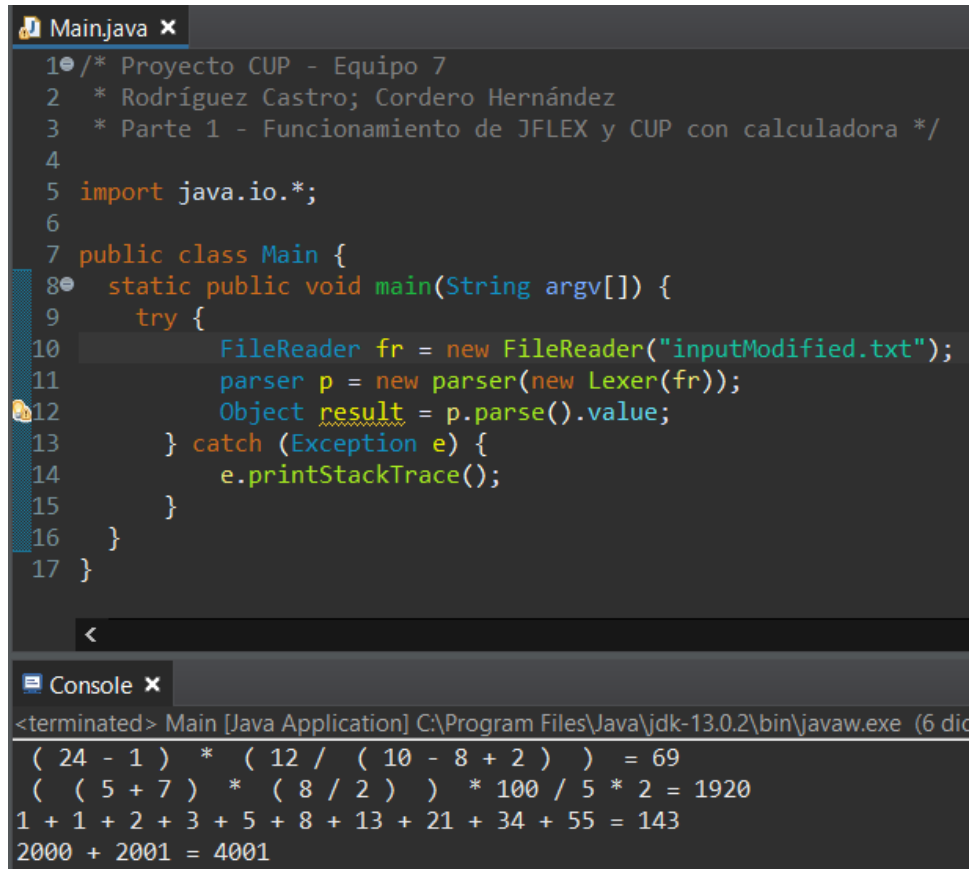
a *
a b c . ,
a b c . a . a , * ,
a b c . a . a , * * ,
a b c . a . a , * , K * ,

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.



```
1  /* Proyecto CUP - Equipo 7
2   * Rodríguez Castro; Cordero Hernández
3   * Parte 1 - Funcionamiento de JFLEX y CUP con calculadora */
4
5  import java.io.*;
6
7  public class Main {
8      static public void main(String argv[]) {
9          try {
10             FileReader fr = new FileReader("inputModified.txt");
11             parser p = new parser(new Lexer(fr));
12             Object result = p.parse().value;
13         } catch (Exception e) {
14             e.printStackTrace();
15         }
16     }
17 }
```

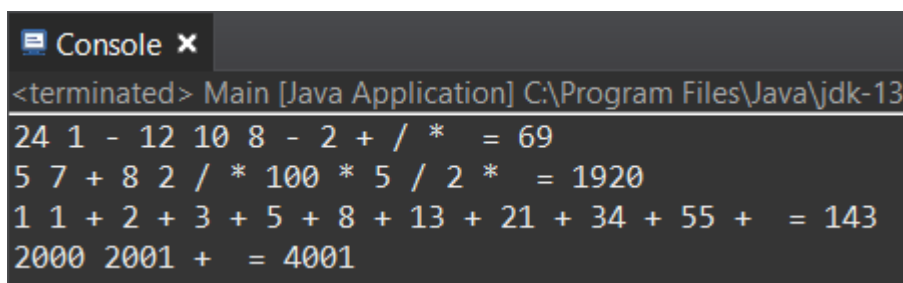
Console

<terminated> Main [Java Application] C:\Program Files\Java\jdk-13.0.2\bin\javaw.exe (6 dic
(24 - 1) * (12 / (10 - 8 + 2)) = 69
((5 + 7) * (8 / 2)) * 100 / 5 * 2 = 1920
1 + 1 + 2 + 3 + 5 + 8 + 13 + 21 + 34 + 55 = 143
2000 + 2001 = 4001

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.



```
Console
<terminated> Main [Java Application] C:\Program Files\Java\jdk-13
24 1 - 12 10 8 - 2 + / * = 69
5 7 + 8 2 / * 100 * 5 / 2 * = 1920
1 1 + 2 + 3 + 5 + 8 + 13 + 21 + 34 + 55 + = 143
2000 2001 + = 4001
```

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

Type a grammar here:

S -> S , E .
S -> E .
E -> E G .
E -> G .
G -> G + .
G -> F .
F -> F * .
F -> H .
H -> (S) .
H -> d .
H -> l .

Example Sentences

• l
• d
• l +
• l *
• d +
• d *
• l + +
• l * +
• l * *
• d + +

Nonterminals				
Symbol	Nullable?	Endable?	First set	Follow set
S		Endable	(, d, l	,,), \$
E		Endable	(, d, l	,, (,), d, l, \$
G		Endable	(, d, l	,, +, (,), d, l, \$
F		Endable	(, d, l	,, +, *, (,), d, l, \$
H		Endable	(, d, l	,, +, *, (,), d, l, \$

LALR(1) The grammar is LALR(1).

[Automaton](#), [Parsing table](#)

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 Associativity 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;

```

g. Captura de pantalla que genere la notación postfija de las expresiones regulares en input.txt

```

a *
a b c . ,
a b c . a . a , * ,
a b c . a . a , * * ,
a b c . a . a , * , K * ,
$
8 * A ,
A 2 . 4 . Z Z . Z . , * * + I . T . U . P + . A . T .
4 + 5 . 1 9 . ,

```

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>.