# Hoja de trabajo: Expresiones regulares y gramáticas

## 1. Expresiones regulares

Escriba una expresión regular (en el formato que usó en jLex) para los siguientes incisos.

1. Los nombres de variable pueden comenzar con una letra minúscula o guión bajo, seguido de cualquier cantidad de minúsculas, mayúsculas, números o guiones bajos.

   **[a-z_][a-zA-Z0-9_]***

2. Los números de punto flotante en mi lenguaje de programación necesitan por lo menos un número antes del punto, o después del punto, no es necesario tener número de ambos lados. Algunos ejemplos de números válidos: 123.  .45  6.7  .0  89.12345

   **[\d]+[\.][\d]+|[\d]+[\.]|[\.][\d]+**

3. Quiero reconocer los espacios en blanco más comunes (espacio, tabulador, newline, retorno de carro).

   **[\t\n\r" "]**

## 2. Ambigüedad en gramáticas

Las siguientes gramáticas son ambiguas. Para cada una genere un par de árboles de parseo donde lo demuestre, y luego reescriba la gramática para que no presente ambigüedad. (No se complique de más, no se preocupe de la precedencia de operaciones)

### Gramática 1

E -> E + E | E * E | N

N -> int | float

### Gramática 2

E -> E X E | id

X -> + | - | * | /

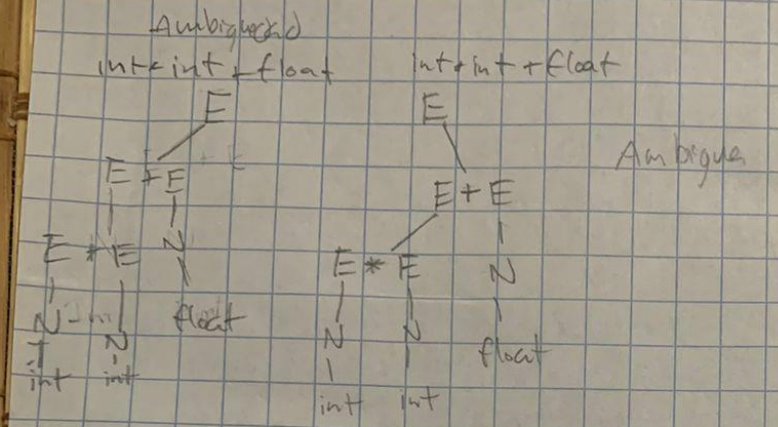
### Gramática 3

A -> A A | 0 | 1

(1)

E → E + E | E * E | N

N → Int | float

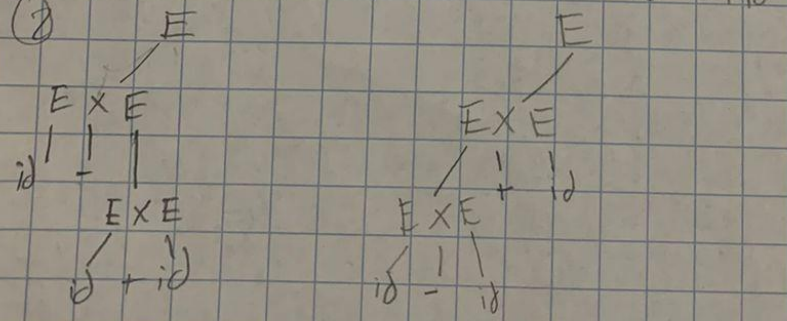



E → E + E
E → E * E
E → N
N → int
N → float

⇒

E → M + E
E → M
M → M * E
E → N
N → int
N → float

(2)



## 3. Recursión por la izquierda

Las siguientes gramáticas presentan recursión por la izquierda, reescríbalas para eliminar el problema.

### Gramática 1

S -> S G | S very | very

G -> good

### Gramática 2

S -> S a | S b | A c | d

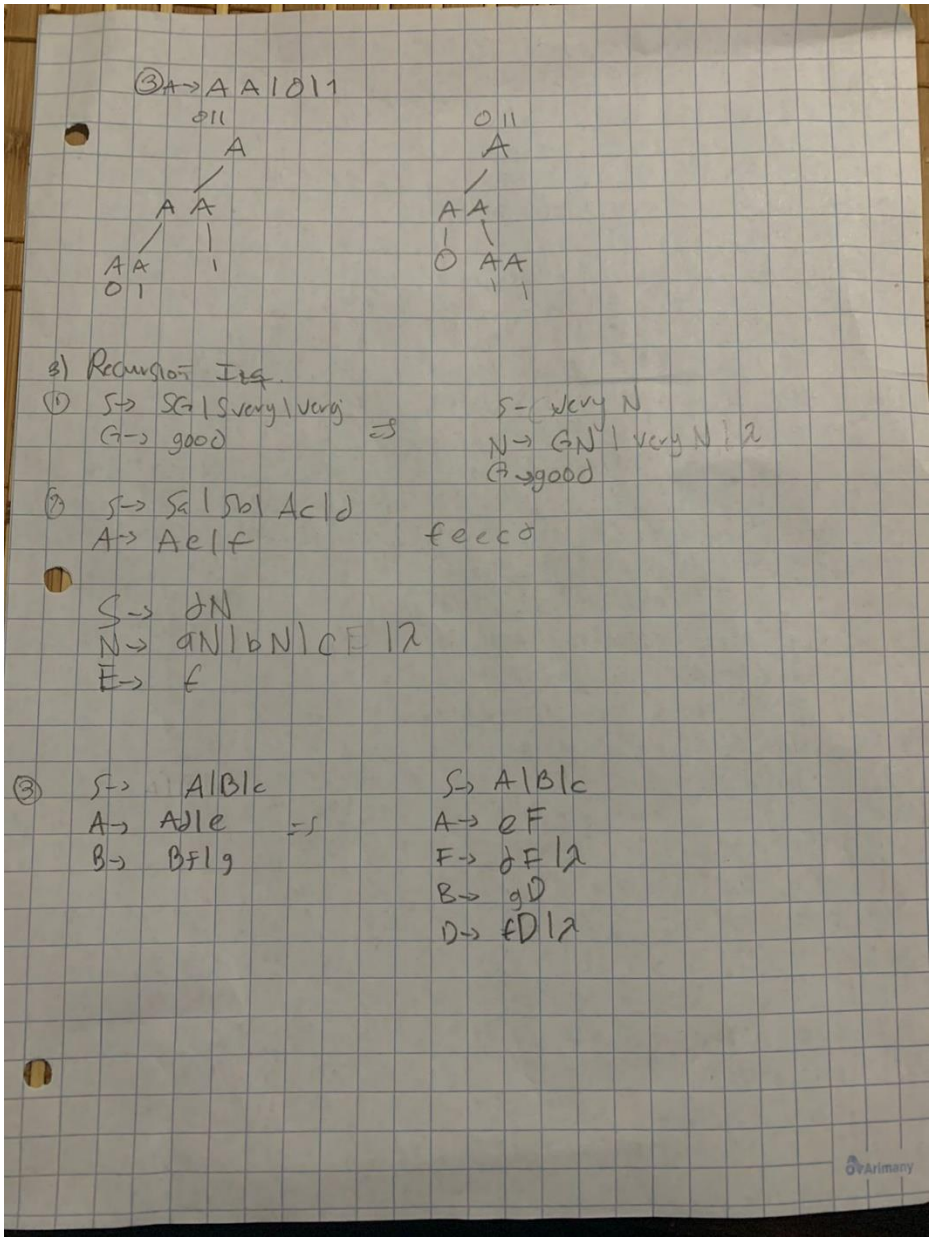
A -> A e | f

### Gramática 3

S -> A | B | c

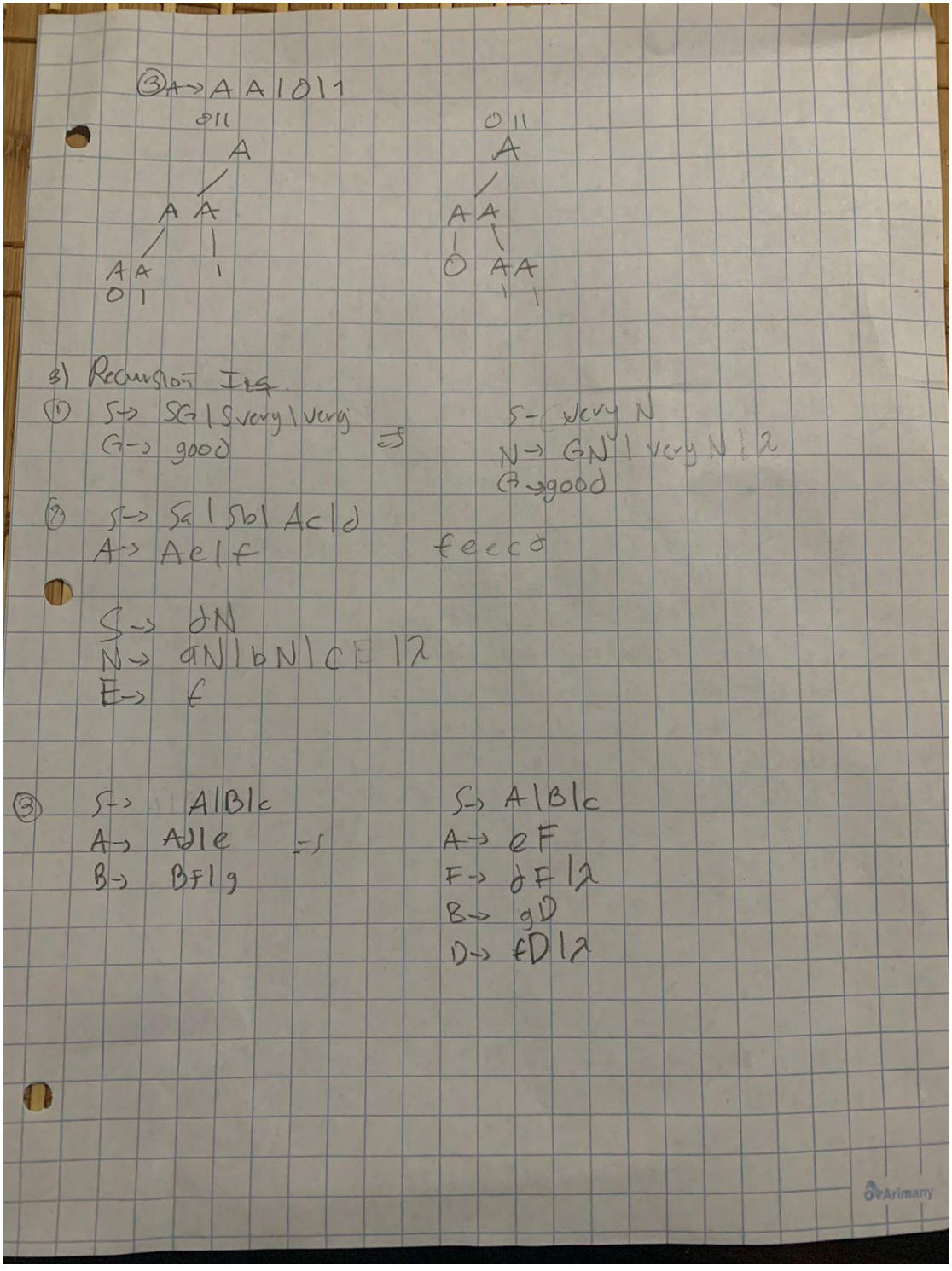
A -> A d | e

B -> B f | g




## 4. Factorizar por la izquierda

Algunas producciones de las siguientes gramáticas tienen elementos en común que pueden ser factorizados. Realice esta factorización.




## Gramática 1

S -> a b C | a D

C -> e | f

D -> g | g h

## Gramática 2

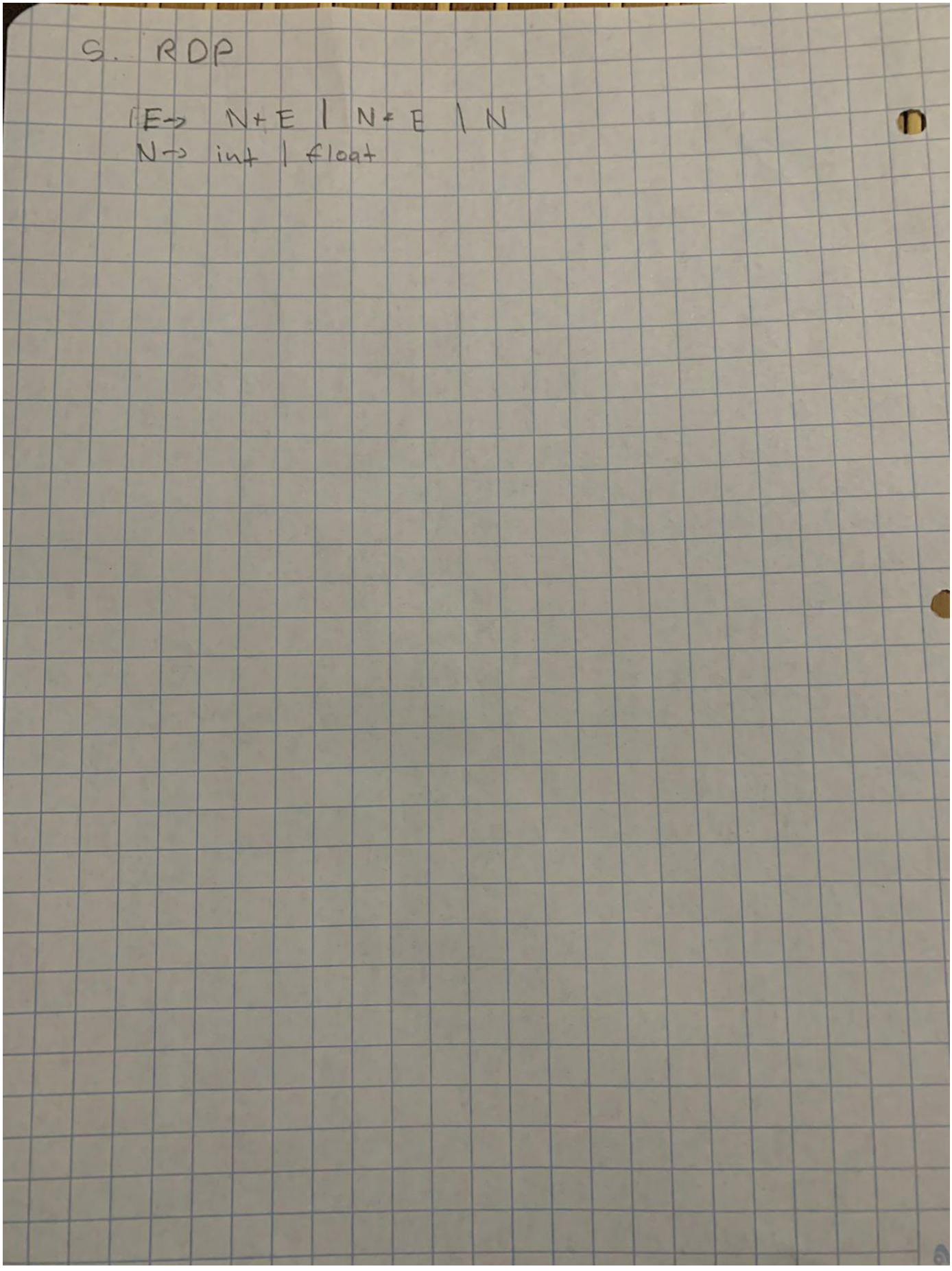
S -> a b c | a b d | x y z | x v w

## Gramática 3

A -> B | c d | c

B -> d e f | d

## 5. Recursive Descent Parser




- Para el siguiente RDP, indique cuál fue la gramática que lo generó.

```
bool term(TOKEN tok) { return *next++ == tok; }

bool E() { TOKEN *save = next;
           return (next = save, E1())
               || (next = save, E2())
               || (next = save, E3()); }

bool N() { TOKEN *save = next;
           return (next = save, N1())
               || (next = save, N2()); }

bool E1() { return N() && term(PLUS) && E(); }

bool E2() { return N() && term(TIMES) && E(); }

bool E3() { return N(); }

bool N1() { return term(INT); }

bool N2() { return term(FLOAT); }
```

- Responda las siguientes preguntas respecto al parser anterior.
  - ¿Por qué las "reglas numeradas" llevan and (&&)? Por ejemplo, `N() && term() && E()` **Eso significa concatenación, que esa regla de producción esta conformada por un conjunto de símbolos. La regla es equivalente a decir N+E**
  - ¿Por qué las "reglas sin numerar" llevan or (||)? Por ejemplo, `return (…) || (…);` **Significa union y este se utiliza por ejemplo en el backtracking de cada símbolo de la regla de producción**
  - ¿Es necesario el `next = save` que aparece antes de llamar a `E1()` o a `N1()`? **Los next = save pueden estar antes o después de la regla de producción.**