União e opcional

- Uma barra (|) em uma expressão regular denota a união dos conjuntos das expressões à esquerda e à direita da barra
  - `[a-zA-Z_][a-zA-Z0-9_]*|[0-9]+` é a união do conjunto denotado por `[a-zA-Z_][a-zA-Z0-9_]*` com o conjunto denotado por `[0-9]*`
- O operador ? denota o conjunto denotado pela expressão que ele modifica, mais a cadeia vazia
  - `[0-9]+([.][0-9]+)?` denota o conjunto de todos as sequências de dígitos, mais o conjunto das sequências de dígitos seguidas por um ponto e outra sequência de dígitos

$[[e]) \cup \{""\} \equiv e$?

$(0-9)+ \equiv (0-9)*$

$([.][0-9]+)? \neq [.][0-9]*$

Precedência

- A precedência dos operadores em uma expressão regular, da menor para a maior, é |, depois concatenação, depois +, * e ?
- Naturalmente, podemos usar parênteses para mudar a precedência quando conveniente
- Na prática, é possível escrever uma especificação léxica sem usar |, ( ) e ?, usando múltiplas regras para a mesma classe de token
- Usar múltiplas regras para alternativas pode deixar a especificação mais legível

Para o mesmo tipo de token

Especificação léxica

- A especificação léxica de uma linguagem é uma sequência de *regras*, onde cada regra é composta de uma expressão regular e um *tipo de token*

- Uma regra diz que se os próximos caracteres presentes na entrada pertencerem ao conjunto denotado pela sua expressão regular, então o próximo token da entrada pertence ao seu tipo

- Para a linguagem de comandos simples, onde os tokens são numerais inteiros, identificadores, +, -, (,), =, ;, print, uma possível especificação léxica é dada no slide seguinte

Comandos simples

```
[0-9]+                      => NUM
[pP][rR][iI][nN][tT]        => PRINT
[a-zA-Z_][a-zA-Z0-9_]* => ID
[+]                         => ‘+’
[-]                         => ‘-’
[(]                         => ‘(‘
[)]                         => ‘)’
=                           => ‘=’
;                           => ‘;’
```

Comandos simples, alternativa

```
[0-9]+                    => NUM
[pP][rR][iI][nN][tT]      => PRINT
[a-zA-Z_][a-zA-Z0-9_]+ => ID
[a-zA-Z_]                 => ID
[+]                       => ‘+’
[-]                       => ‘-’
[(]                       => ‘(‘
[)]                       => ‘)’
=                         => ‘=’
;                         => ‘;’
```

Comandos simples, alternativa 2

```
[0-9]+                          => NUM
[pP][rR][iI][nN][tT]            => PRINT
[a-zA-Z_][a-zA-Z0-9_]+|[a-zA-Z_] => ID
[+]                             => ‘+’
[-]                             => ‘-’
[(]                             => ‘(‘
[)]                             => ‘)’
=                               => ‘=’
;                               => ‘;’
```

Um fragmento de Java

```
&&                          => E_LOGICO
[|][|]                      => OU_LOGICO
[+]                         => '+'
[+][+]                      => INC
/                           => '/'
[.]                         => '.'
while                       => WHILE
if                          => IF
for                         => FOR
else                        => ELSE
[a-zA-Z]                    => ID
[a-zA-Z_][a-zA-Z0-9_]+      => ID
[0-9]+                      => NUM
[0-9]+[.][0-9]+             => NUM
[0-9]+[.]                   => NUM
[.][0-9]+                   => NUM
["]["]                      => STRING
["][^"\n]+["]               => STRING
```

Ambiguidade na especificação

- Uma especificação mais complexa como a de Java é naturalmente ambígua
  - Uma entrada `123.4` pode ser um token NUM (“123.4”), dois tokens NUM (“123” e “.4”), um token NUM seguido de um ‘.’ seguido de outro NUM (“123”, “.”, “4”), ou variações disso (“1”, “23”, “.4”)
  - Uma entrada `fora` pode ser um token ID (“fora”), ou um token FOR e um ID (“for”, “a”)
  - `while` pode ser tanto um ID quanto um token WHILE
- Precisamos de regras para remoção da ambiguidade

Removendo ambiguidade

- Caso mais de uma regra consiga classificar os próximos caracteres da entrada, dá-se preferência aquela que consegue classificar **o maior número de caracteres**
  - Ou seja, `123.4` é um único token NUM, e `fora` é um token ID
- Se ainda assim existem várias regras que classificam o mesmo número de caracteres, dá-se preferência à que **vem primeiro**
  - Logo, `while` seria classificado como WHILE

Comandos simples – errado

```
[0-9]+                      => NUM
[a-zA-Z_][a-zA-Z0-9_]* => ID
[pP][rR][iI][nN][tT]   => PRINT
[+]                         => ‘+’
[-]                         => ‘-’
[(]                         => ‘(‘
[)]                         => ‘)’
=                           => ‘=’
;                           => ‘;’
```