

# Compiladores - Análise Léxica

---

Fabio Mascarenhas – 2015.2

<http://www.dcc.ufrj.br/~fabiom/comp>

# Introdução

---

- Primeiro passo do front-end: reconhecer *tokens*
- Tokens são as palavras do programa
- O analisador léxico transforma o programa de uma sequência de caracteres sem nenhuma estrutura para uma sequência de tokens

```
if (x == y) then  
    z = 1;  
else  
    z = 2;
```

```
if | |x| |==| |y| |then|\n |z| |=| |1|;|\n else|\n |z| |=| |2|;|<EOF>
```

# Tipo do token

---

- Em português:
  - substantivo, verbo, adjetivo...
- Em uma linguagem de programação:
  - identificador, numeral, if, while, (, ;, *identação*, ...

# Tipo do token

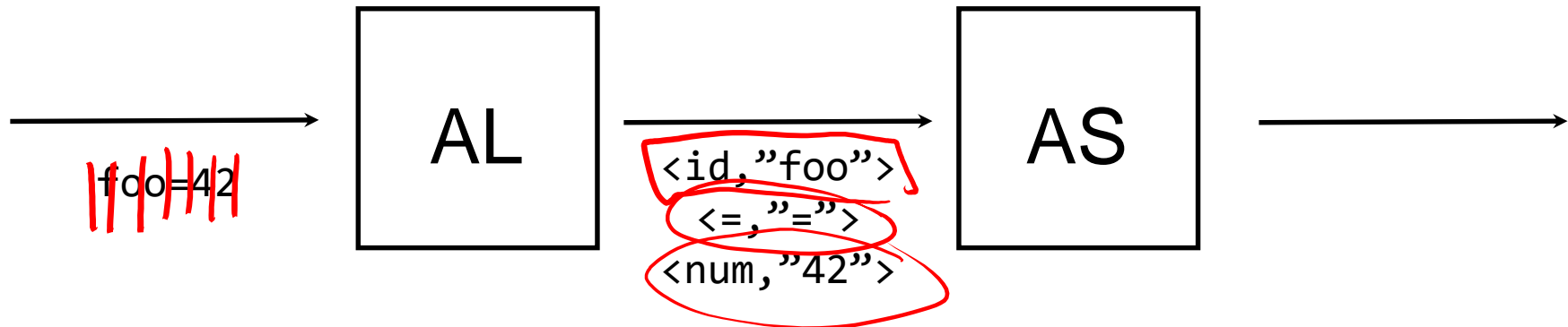
---

- Cada tipo corresponde a um *conjunto de strings*
- Identificador: *strings de letras ou dígitos, começadas por letra*
- Numeral: *strings de dígitos*
- Espaço em branco: *uma string de brancos, quebras de linha, tabs, ou comentários*
- while: *a string while*

# Análise léxica

---

- Classificar substrings do programa de acordo com seu tipo
- Fornecer esses tokens (par tipo e substring) ao analisador sintático



# Exemplo

---

- Para o código abaixo, conte quantos tokens de cada tipo ele tem

```
x = 0;\nwhile (x < 10) {\n\tx++;\n}\n
```

Tipos: id, espaço, num, while, outros

id    |||  
espaço    ||| ||| ||| |||  
outros    ||| ( ||| |||  
num    ||

while |

```
x = 0;  
while (x < 10) {  
    x++;  
}
```

# Exemplo

---

- Para o código abaixo, conte quantos tokens de cada tipo ele tem

```
x = 0;\nwhile (x < 10) {\n\tx++;\n}\n
```

Tipos: id (3), espaço (10), num (2), while (1), outros (9)

# Ambiguidade

---

- A análise léxica de linguagens modernas é bem simples, mas historicamente esse não é o caso
- Em FORTRAN, espaços em branco *dentro de um token* também são ignorados
  - VAR1 e VAR 1 são o mesmo token
  - D05I=1,25 são 7 tokens: “DO”, “5”, “I”, “=”, “1”, “,”, “25”
  - Já D05I=1.25 são 3 tokens: “D05I”, “=”, “1.25”



# Ambiguidade

---

- As palavras-chave de PL/1 não são reservadas

*PR* *IF* *PR* *IF* = *IF* *PR* *IF* = *IF*  
• IF ELSE THEN THEN = ELSE; ELSE ELSE = THEN

- Mas mesmo linguagens modernas têm ambiguidades léxicas

• ==, ++, +=

- Templates C++/Generics Java: List<List<Foo>> vs foo >> 2;

*IF < IF < IF RESOLVE*

*RESOLVER  
DEPUIS*

- O analisador léxico precisa manter um “lookahead” para saber onde um token começa e outro termina


# Linguagens regulares

---

- Um tipo de token é um conjunto de strings
- Outro nome para conjunto de strings é *linguagem*
- Geralmente os conjuntos de strings que caracterizam os tipos de tokens de linguagens de programação são *linguagens regulares*
- Em linguagens formais, uma *linguagem regular* é qualquer conjunto de strings que pode ser expresso usando uma *expressão regular*
- Logo, o fato dos tipos de tokens serem linguagens regulares dá uma notação conveniente para especificarmos como classificar os tokens!

# Expressões regulares

---

- Assim como uma expressão aritmética denota um número (por exemplo, “2+3\*4” denota o número 14, uma expressão regular denota uma linguagem regular
  - Por exemplo  `a0+` denota a linguagem { “a0”, “a00”, “a000”, ... }
- Vamos explorar expressões regulares usando a função `Tex.RE.findAll`, que recebe uma expressão regular e uma string e retorna todas as ocorrências daquela expressão regular na string
  - `findAll(“a0+”, “a0 fooa000bar a005”)` => [“a0”, “a000”, “a00”]

# Caracteres e classes

---

- *Caracteres e classes de caracteres* são o tipo mais simples de expressão regular
- Denotam conjuntos de cadeias de um único caractere
- A expressão  $a$  denota o conjunto  $\{ "a" \}$ , a expressão  $x$  o conjunto  $\{ "x" \}$
- A expressão  $.$  é especial e denota o *conjunto alfabeto* (conjunto de todos os caracteres)
- Uma *classe*  $[abx]$  denota o conjunto  $\{ "a", "b", "x" \}$
- Uma classe  $[ab-fx]$  denota  $\{ "a", "b", "c", "d", "e", "f", "x" \}$
- Uma classe  $[\wedge ab-fx]$  denota o *conjunto complemento* da classe  $"[ab-fx]"$  em relação ao alfabeto

exceto \n

$\wedge \rightarrow \{ " \backslash " \}$       $\wedge . \rightarrow \{ " . " \}$

# Concatenação ou justaposição

---

- A concatenação ou justaposição de expressões regulares denota um conjunto com cadeias de vários caracteres, onde cada caractere da cadeia vem de uma das expressões concatenadas

- $[a-z][0-9]$  denota o conjunto { "a0", "a1", ..., "a9", "b0", ..., "b9", ..., "z9" }

- $while$  denota o conjunto { "while" }

$\{ 'w' \} \{ 'h' \} \{ '.' \} \{ 'l' \} \{ 'e' \}$

- $[wW][hH][iI][lL][eE]$  denota o conjunto { "while", "While", "wHile", "WHile", ... }

- ... denota o conjunto de todas as cadeias de três caracteres (incluindo espaços!)

$\{ ' ' \} \{ ' ' \} \{ ' ' \}$

# Repetição

---

- O operador + denota a *repetição* de um caractere ou classe de caracteres
  - $[a-z]^+$  denota o conjunto { "a", "aa", "aaa", ..., "b", "bb", ..., "aba", ... }, ou seja, cadeias formadas de caracteres entre a e z
  - $[a-z][0-9]^+$  denota o conjunto { "a0", "a123", "d25", ... }, ou seja, cadeias formadas por um caractere de a a z seguidas por um ou mais dígitos
- O operador \* é uma repetição que permite *zero* caracteres ao invés de ao menos 1
  - $[a-z][0-9]^*$  denota o conjunto acima, mais o conjunto { "a", "b", ... "z" }
  - $\backslash"[^\\"]*\backslash"$  denota o conjunto de cadeias de quaisquer caracteres entre aspas duplas, exceto as próprias aspas duplas, e inclui a cadeia ""

{ "", "c", ". ", "43", ... }

# 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
- $[0-9]^+([\cdot][0-9]^+)?$  denota o conjunto de todas as sequências de dígitos, mais o conjunto das sequências de dígitos seguidas por um ponto e outra sequência de dígitos

$[0-9]^+ | [0-9]^+[\cdot][0-9]^+$

# 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



# 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

---

<code>[0-9]+</code>	<code>=&gt;</code>	<code>NUM</code>
<code>[pP][rR][iI][nN][tT]</code>	<code>=&gt;</code>	<code>PRINT</code>
<code>[a-zA-Z_][a-zA-Z0-9_]*</code>	<code>=&gt;</code>	<code>ID</code>
<code>[+]</code>	<code>=&gt;</code>	<code>'+'</code>
<code>[-]</code>	<code>=&gt;</code>	<code>'-'</code>
<code>[ ( ]</code>	<code>=&gt;</code>	<code>'('</code>
<code>[ ) ]</code>	<code>=&gt;</code>	<code>')'</code>
<code>=</code>	<code>=&gt;</code>	<code>'='</code>
<code>;</code>	<code>=&gt;</code>	<code> ';' </code>

# 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  
*ou mesmo comandos simples*
- 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  
*fora*
- 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

---

<code>[0-9]+</code>	<code>=&gt;</code>	<code>NUM</code>
<code>[a-zA-Z_][a-zA-Z0-9_]*</code>	<code>=&gt;</code>	<code>ID</code>
<code>[pP][rR][iI][nN][tT]</code>	<code>=&gt;</code>	<code>PRINT</code>
<code>[+]</code>	<code>=&gt;</code>	<code>'+'</code>
<code>[-]</code>	<code>=&gt;</code>	<code>'-'</code>
<code>[ ( ]</code>	<code>=&gt;</code>	<code>'('</code>
<code>[ ) ]</code>	<code>=&gt;</code>	<code>)'</code>
<code>=</code>	<code>=&gt;</code>	<code>'='</code>
<code>;</code>	<code>=&gt;</code>	<code>';'</code>