FUNDAMENTOS TEÓRICOS DA COMPUTAÇÃO

-- APLICAÇÃO DE EXPRESSÕES REGULARES --

Análise Léxica

Padrões, lexemas e tokens

Padrão (Expressão Regular)	Lexema	Token
`digito+`	`547`	`<t_num, 547>`
`letra ( letra | dígito )*`	`x1`	`<t_id, 1>`
`while`	`while`	`<t_while>`

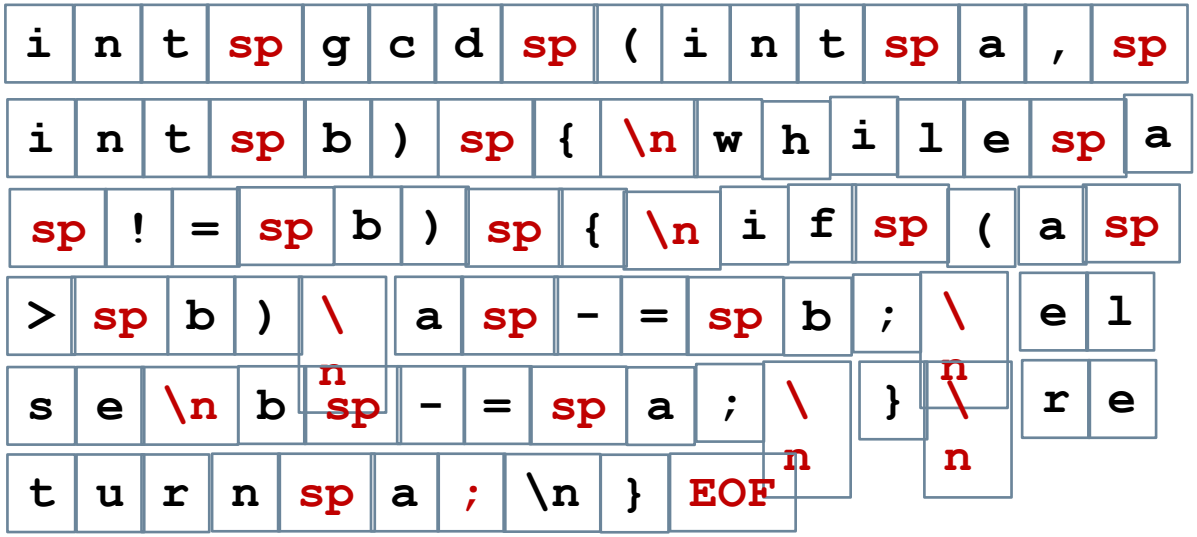
Padrões, lexemas e tokens

Seja a cadeia de entrada: `x = y * 2;`

Lexema	Token
x	<t_id, 1>
=	<t_atrib>
y	<t_id, 2>
*	<t_mult>
2	<t_num, 3>
;	<t_end_com>

Exemplos de lexemas e tokens

```
int gcd (int a, int b) {
  while (a != b) {
    if (a > b)
      a -= b;
    else
      b -= a;
  }
  return a;
}
```



Exemplos de lexemas e tokens

```
int gcd (int a, int b) {
  while (a != b) {
    if (a > b)
      a -= b;
    else
      b -= a;
  }
  return a;
}
```

int	gcd	(	int	a	,	int	b	)	{	while
(	a	!=	b	)	{	if	(	a	>	b	)	a	-=	b
;	else	b	-=	a	;	}	return	a	;	}

Expressões Regulares e Autômatos Finitos

Metacarcteres

Expressões Regulares identificam **padrões** em cadeias de caracteres. Elas **não validam dados**, apenas verificam se um texto está em um determinado **padrão**. Os padrões são definidos por **metacarateres**

Metacaractere	Descrição
`x`	o caractere 'x'
`.`	todos os caracteres (curinga) exceto '\n'
`[xyz]`	'x' ou 'y' ou 'z'
`[x-z]`	todos os caracteres entre 'x' e 'z'
`[^x-z]`	todos os caracteres exceto os caracteres entre 'x' e 'z'
`r*`	a expressão regular r zero ou mais vezes
`r+`	a expressão regular r uma ou mais vezes
`r?`	a expressão regular r é opcional
`r{n}`	a expressão regular r n vezes

Metacarcteres

Metacaractere	Descrição
`r{n,}`	a expressão regular r n ou mais vezes
`r{n,m}`	a expressão regular r entre n a m vezes
`(r)`	a expressão regular r; necessário para contornar regras de precedência
`rs`	concatenação das expressões regulares r e s
`r|s`	ou a expressão regular r ou a expressão regular s
`r/s`	a expressão regular r, mas só se for seguida da expressão regular s
`^r`	a expressão regular r, mas só se for no início da linha
`r$`	a expressão regular r, mas só se for no fim da linha

Exemplos de Expressões Regulares

ER	Descrição
`-?[0-9]+`	Inteiros que podem ser negativos
`foo|bar*`	Casa com a string ‘foo’ ou ‘ba’ seguida de 0 ou mais ‘r’
`foo|(bar)*`	Casa com foo ou zero ou mais bar
`(foo|bar)*`	Casa com zero ou mais foo ou bar
`a{3}`	Casa com 'aaa'
`a{3,4}`	Casa com 'aaa' ou 'aaaa'
`(ba){3}`	Casa com 'bababa'
`[A-Za-z] [A-Za-z0-9]*`	Casa com um identificador
`[a-z][a-z]`	Casa com duas letras minúsculas

Testes de Expressões Regulares

Qual seria a expressão regular que casaria com as três strings abaixo?

`abcdefg`
`abcde`
`abc`

Resposta: **`abc(de)?(fg)?`**

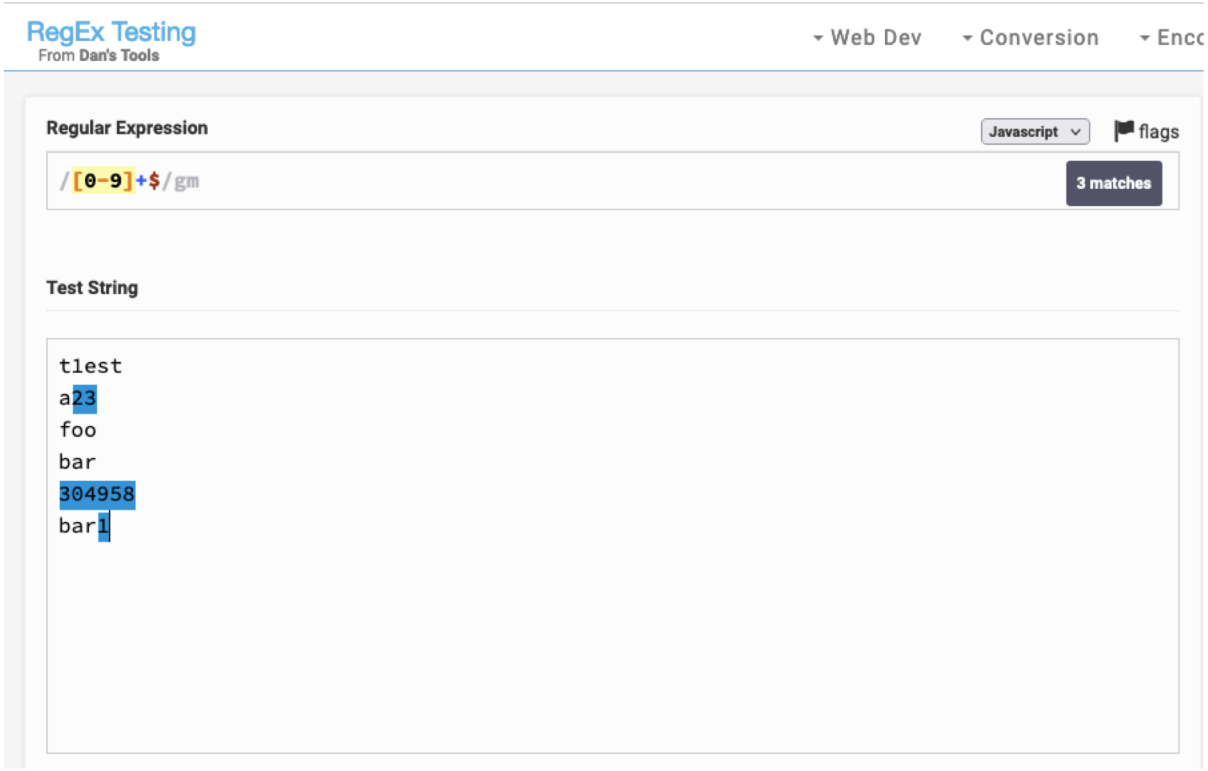
Testes de Expressões Regulares

Qual seria a expressão regular que casaria com as três strings abaixo?

```
abc123xyz
define "123"
var g = 123;
```

Resposta: `.123.`

Testes de Expressões Regulares



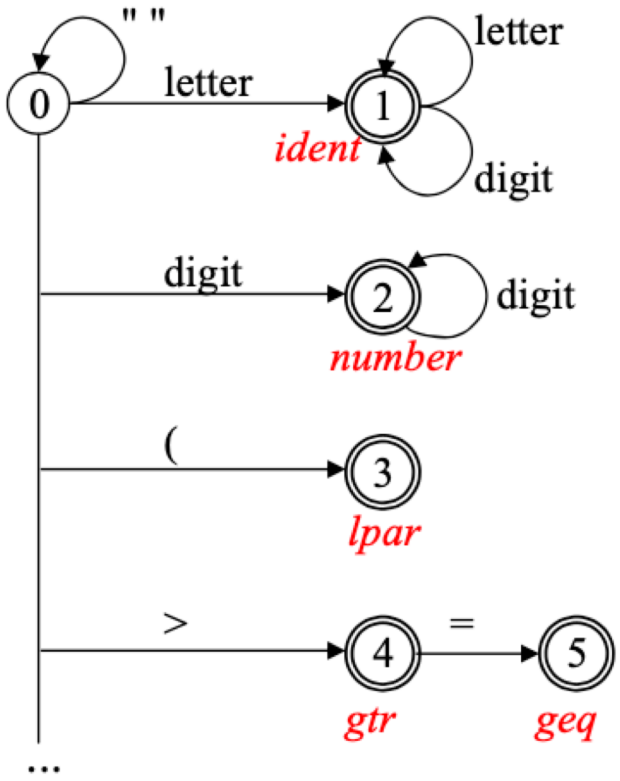
https://www.regextester.com/

Scanner como Autômato Finito Determinístico (AFD)

O scanner é um grande AFD

Cada vez que um token é reconhecido, estado volta para 0

Quando minimizado, é o mais eficiente possível



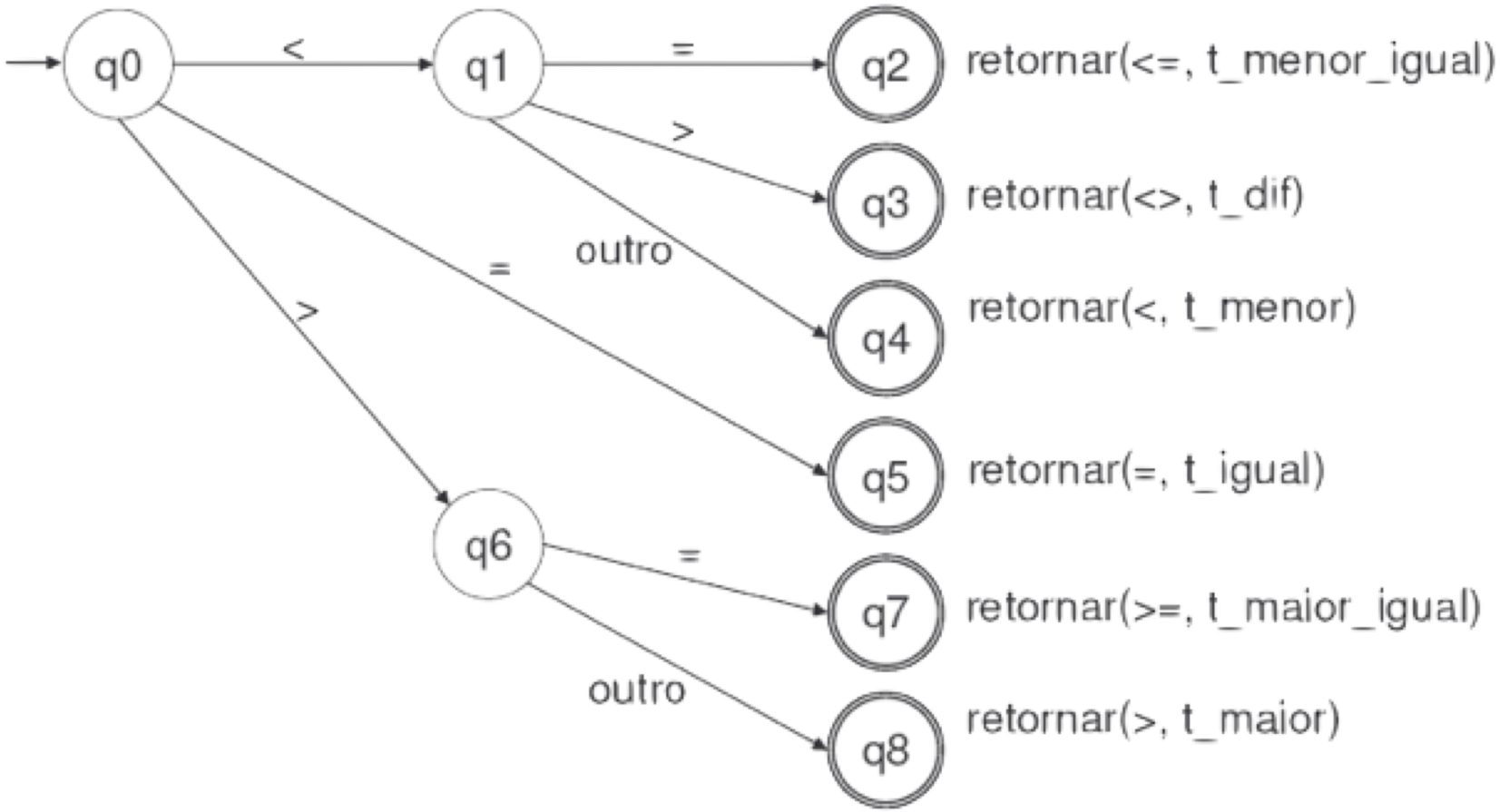
m	a	x		>	=		3	0

s0 —m→ s1 —a→ s1 —x→ s1 ident

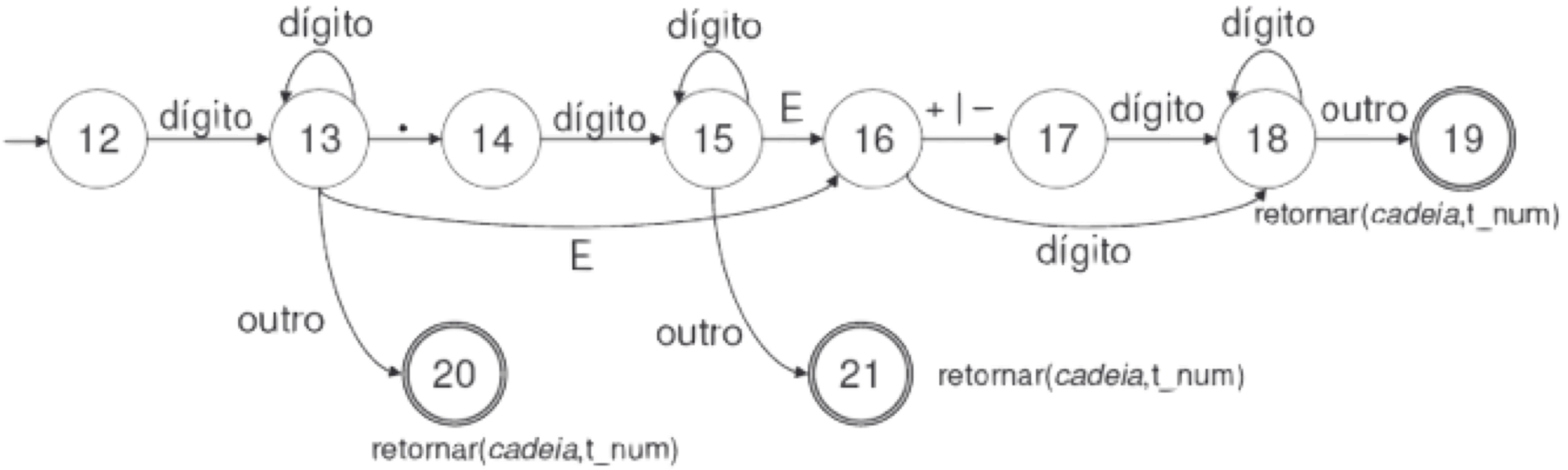
s0 —" "→ s0 —>→ s4 —=→ s5 geq

s0 —" "→ s0 —3→ s2 —0→ s2 number

Autômato para operadores relacionais



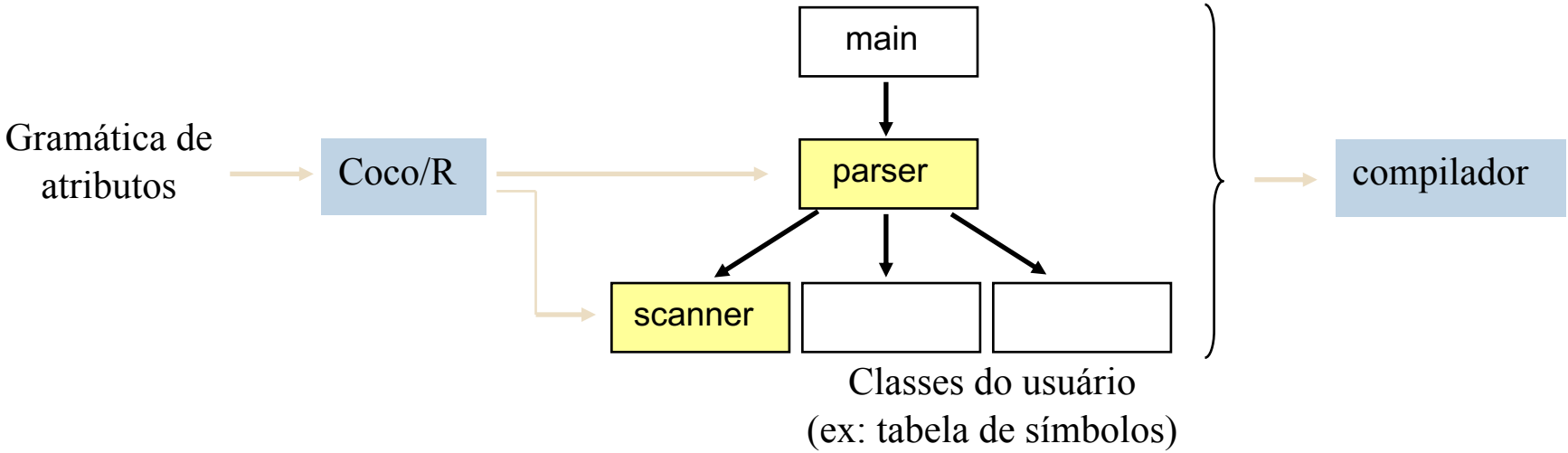
Autômato para números



COCO/R

Coco/R - Compiler Compiler / Recursive Descent

Como funciona



Exemplo **muito** simples

Parser para as alternativas

banana amarela

laranja

Gramática...

Simples = “banana" “amarela" | “laranja".

Descrição Coco/R

```
COMPILER Simples
PRODUCTIONS
   Simples = “banana" “amarela" | “laranja".
END Simples.
```

Arquivo *simples.atg*

Rodamos Coco/R para gerar scanner e parser

```
> java -jar ../Coco.jar  -frames .. Simples.atg
Coco/R (Aug 22, 2006)
checking
parser + scanner generated
0 errors detected
```

Exemplo **muito** simples

Escrevemos o código principal

```
import java.io.*;

public class Simples
{
    public static void main(String[] arg) {
        Scanner scanner = new Scanner(arg[0]);
        Parser parser = new Parser(scanner);
        parser.Parse();
        System.out.println(parser.errors.count + " errors detected");
    }
}
```

O que faz?

- Cria scanner
- Cria parser
- Inicia o parser
- Reporta número de erros

Compila tudo ...

```
> javac Simples.java Scanner.java Parser.java
```

... E roda

```
> java Simples exemplo.txt
0 errors detected
```

Arquivo *exemplo.txt*

```
banana amarela
```

numero.atg

Um scanner que reconhece números que estejam no início de uma linha

```
COMPILER Numero

CHARACTERS
    digito = '0'..'9'.
    outro = ANY - digito .

TOKENS
    numero = digito { digito } .
    outros = outro {outro}.

IGNORE '\t' + '\r'

PRODUCTIONS
    Numero  = numero [outros] .
END Numero.
```

Numero.java

```
import java.io.*;
public class Numero {
  public static void main(String[] arg) {
    Scanner scanner = new Scanner(arg[0]);
    Parser parser = new Parser(scanner);
    parser.Parse();
    System.out.println(parser.errors.count +
      " errors detected");
  }
}
```

Compilação e execução

```
$ java -jar ../Coco.jar -frames .. numero.atg
Coco/R (Apr 19, 2011)
checking
parser + scanner generated
0 errors detected

$ javac *.java

$ java Numero exp1.txt
0 errors detected

$ java Numero exp3-erro.txt
-- line 1 col 1: numero expected
1 errors detected
```