Construção de compiladores

Profs. Mário César San Felice (e Helena Caseli,
Murilo Naldi, Daniel Lucrédio)
Departamento de Computação - UFSCar
1º semestre / 2018
Tópico 2 - Análise Léxica

Análise léxica

Contexto

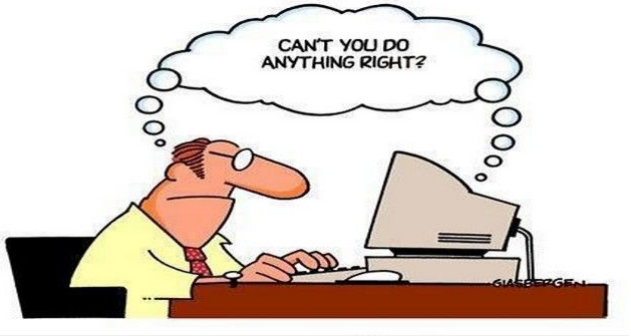
- No filme Matrix, porque os agentes conversam usando linguagem humana?
  - Sistemas não trocam informação deste jeito!
    - **Linguagem humana é pouco eficiente**

- No filme, eles também conversam através dos "plugues" no ouvido
  - Apenas quando **nós precisamos compreender** é que eles falam nossa linguagem entre si

Contexto

- Em compiladores:
  - **Computador precisa entender um programa** com
    - Instruções (cálculos, E/S)
    - Dados
    - Comentários

- É uma **conversa**!
  - Unilateral (ou quase) - homem passando informações para máquina



Fonte: https://www.interaction-design.org/literature/article/a-brief-history-of-human-computer-interaction

Contexto

- Sendo uma conversa (mesmo que unilateral)
  - **Programador precisa compreender a linguagem**
    - Para poder formular e raciocinar corretamente

- Humanos compreendem linguagem humana
  - Linguagem humana tem:

Vocabulário + Gramática

- Nomes das coisas
- É o que mais facilmente emerge na consciência dos locutores
- Quem tem bebês sabe como eles aprendem a falar

- Ações
- Composição
- Conceitos complexos

Contexto

- Sendo uma conversa
  - **O computador também precisa "entender"**
    - Uma "fala" é só uma sequência de caracteres

- As pessoas não pensam muito nisso
  - Parece "óbvio" seguir nosso modelo
  - Mas **poderíamos implementar uma máquina para interpretar essa cadeia diretamente**
    - Uma máquina de Turing, por exemplo

Contexto

- Seguir o modelo natural é muito útil, pois:

  1. **Humanos entendem melhor** os programas que seguem nosso "modelo" de linguagens:
     - Palavras e espaços

  2. **A implementação é mais simples**
     - Léxico vs sintático: Problemas diferentes exigem soluções diferentes
       - Léxico: reconhecer palavras
       - Sintático: reconhecer frases

Contexto

3. O trabalho do analisador sintático é facilitado
   - Opção 1 (tudo junto)
     - Na análise sintática, seria preciso lidar com espaços, comentários, nomes de tokens e o problema da sensibilidade ao contexto
   - Opção 2 (separar léxico do sintático):
     - Léxico descarta espaços e comentários; substitui nomes, valores e constantes; guarda informações de contexto
     - Sintático pode tratar **cada CLASSE de lexemas como um único símbolo terminal**.
       - A gramática fica muito mais simples.

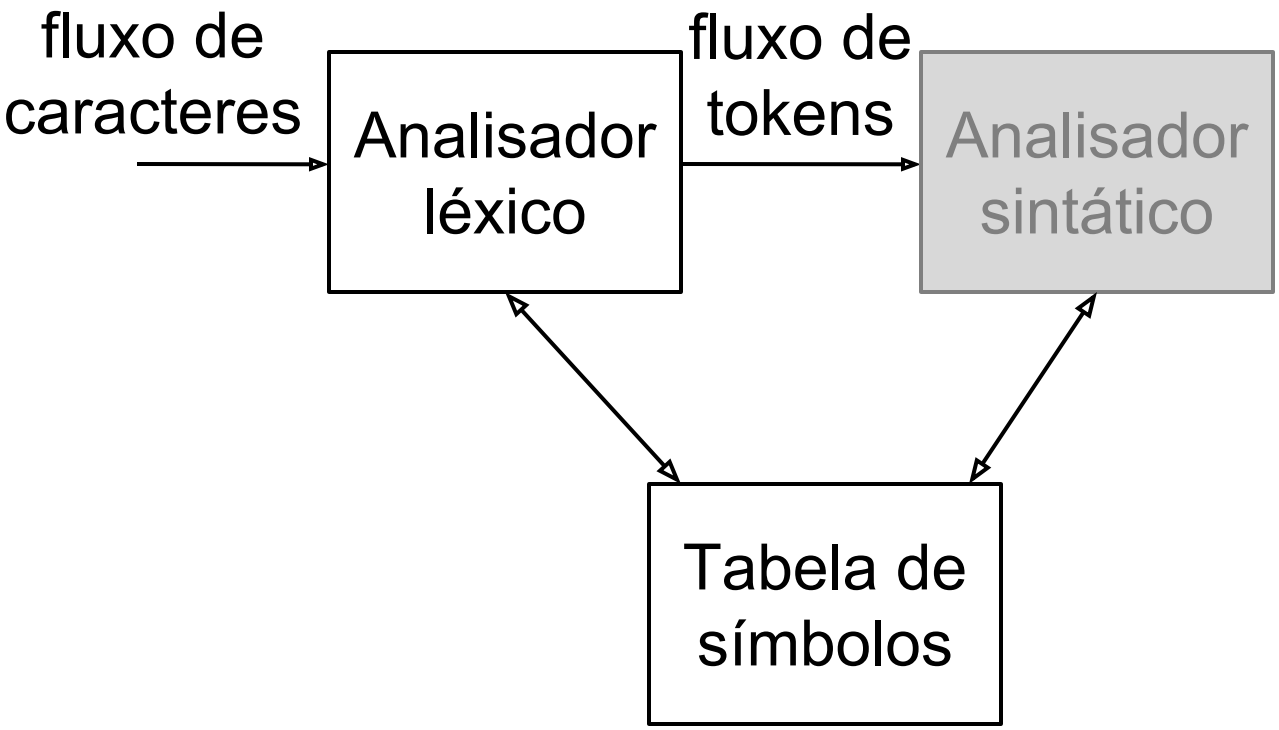
Contexto

4. Eficiência
   - É possível **otimizar tarefas de leitura**
     - Como criar um buffer de entrada no léxico

5. Portabilidade
   - Peculiaridades de leitura não “poluem” o parser
   - Por exemplo:
     - Manter o número de linha para reportar erros
     - Isso pode ser tratado no léxico
     - Portanto, **o parser fica mais independente**

O analisador léxico

Contexto



Obs: dentro da análise (front-end) do processo análise-síntese

Contexto

- Outras tarefas do analisador léxico
  - Remover comentários
  - Remover espaços em branco
    - incluindo tabulação, enter, fim de linha
  - Correlacionar mensagens de erro com o programa fonte (ex: número de linha, coluna)

Lexema vs padrão vs token

- Lexema:
  - Sequência de caracteres no programa fonte
- Por exemplo:

```
if(var1 > 37) {
    outraVar += 54;
    System.out.println(“ok”);
}
```

- Lexemas neste trecho:

if	(	var1	>	37
)	{	outraVar	+=	54
;	System	.	out	.
println	(	“ok”	)	;
}				

Lexema vs padrão vs token

- Padrão:
  - Caracteriza CLASSES de lexemas
- Por exemplo:

```
if(var1 > 37) {
    outraVar += 54;
    System.out.println("ok");
}
```

- Classes de lexemas neste trecho:

identificadores	var1, outraVar, System, out, println
constantes (numéricas)	37, 54
constantes (cadeias)	"ok"
operadores	>, +=
palavras-chave	if
...	

Lexema vs padrão vs token

- Um padrão é utilizado pelo analisador léxico para:
  - Reconhecer lexemas
  - E classificá-los

- Por exemplo:

Classe	**Lexemas**	**Padrão**
identificadores	var1, outraVar, System, out, println	Cadeia de caracteres começando com letra
constantes (numéricas)	37, 54	Sequência de dígitos
constantes (cadeias)	“ok”	Cadeia de caracteres envolta por aspas
operadores	>, +=	O próprio lexema
palavras-chave	if	O próprio lexema
...		

Lexema vs padrão vs token

- Token:
  - Unidade léxica correspondente a um lexema
  - Estrutura de dados:

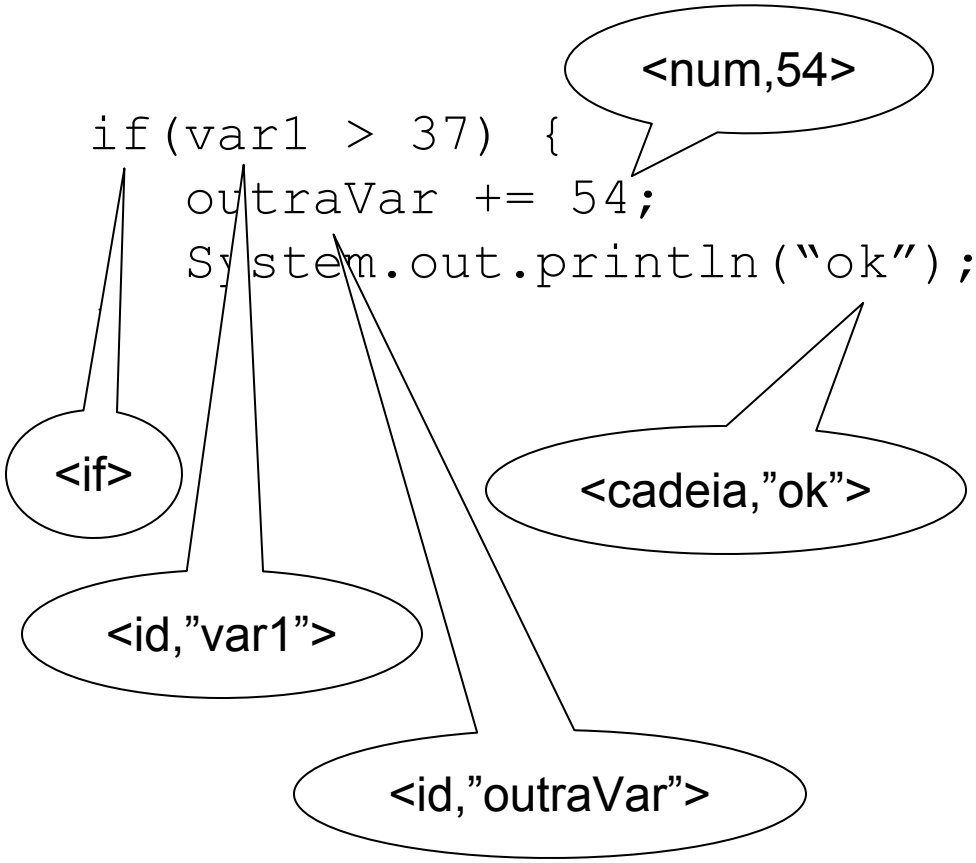
```
class Token {
   TipoToken tipo;
   String valor;
}
```

<tipo,valor>

- Tipo do token → usado pelo analisador sintático
- Valor → O próprio lexema ou outras informações:
  - Valor numérico, se for uma constante
  - Ponteiro para a tabela de símbolos

Lexema vs padrão vs token

- Alguns tokens reconhecidos neste exemplo:



Classe	Padrão	Sigla
identificadores	Cadeia de caracteres começando com letra	id
constantes (numéricas)	Sequência de dígitos	num
constantes (cadeias)	Cadeia de caracteres envolta por aspas	cadeia
operadores	O próprio lexema	op
palavras-chave	O próprio lexema	o próprio lexema

Exemplo: linguagem ALGUMA

- ALGUMA
  - ALGoritmos Usados para Mero Aprendizado
- Usaremos essa linguagem nos exemplos a seguir

- É uma linguagem de programação simples com:
  - Declaração de variáveis (inteiras e reais)
  - Expressões aritméticas (+,-,*,/)
  - Expressões relacionais (>, >=, <=, <, =, <>)
  - Expressões lógicas (E, OU)
  - Condicional (SE-ENTÃO-SENÃO)
  - Repetição (ENQUANTO)

Exemplo: linguagem ALGUMA

```
:DECLARACOES
argumento:INT
fatorial:INT

:ALGORITMO
% Calcula o fatorial de um número inteiro
LER argumento
ATRIBUIR argumento A fatorial
SE argumento = 0 ENTAO ATRIBUIR 1 A fatorial
ENQUANTO argumento > 1
   INICIO
      ATRIBUIR fatorial * (argumento - 1) A fatorial
      ATRIBUIR argumento - 1 A argumento
   FIM
IMPRIMIR fatorial
```

Exemplo: linguagem ALGUMA

Padrão	**Tipo de lexema**	**Sigla**
DECLARAÇÕES, ALGORITMO, INTEIRO, REAL, ATRIBUIR, A, LER, IMPRIMIR, SE, ENTAO, ENQUANTO, INICIO, FIM	Palavras-chave	3 primeiras letras
*, /, +, -	Operadores aritméticos	OpArit
<, <=, >=, >, =, <>	Operadores relacionais	OpRel
E, OU	Operadores booleanos	OpBool
:	Delimitador	Delim
(,)	Parêntesis	AP / FP
Sequências de letras ou números que começam com letra	VARIÁVEL	Var
Sequências de dígitos (sem vírgula)	NÚMERO INTEIRO	NumI
Sequências de dígitos (com vírgula)	NÚMERO REAL	NumR
Sequências de caracteres envolta por aspas	CADEIA	Str

Exercício

Identifique os tokens do programa abaixo, conforme os padrões da linguagem ALGUMA. No campo “valor”, armazene o lexema se necessário

```
:DECLARACOES
argumento:INTEIRO
fatorial:INTEIRO
:ALGORITMO
% Calcula o fatorial de um número inteiro
LER argumento
ATRIBUIR argumento A fatorial
SE argumento = 0 ENTAO
 ATRIBUIR 1 A fatorial
ENQUANTO argumento > 1
 INICIO
  ATRIBUIR fatorial * (argumento - 1)
           A fatorial
  ATRIBUIR argumento - 1 A argumento
 FIM
IMPRIMIR fatorial
```

Padrão	Sigla
DECLARAÇÕES, ALGORITMO, INTEIRO, REAL, ATRIBUIR, A, LER, IMPRIMIR, SE, ENTAO, ENQUANTO, INICIO, FIM	3 primeiras letras
*, /, +, -	OpArit
<, <=, >=, >, =, <>	OpRel
E, OU	OpBool
:	Delim
(,)	AP / FP
Seq. de letras ou números que começam com letra minúscula	Var
Sequências de dígitos (sem vírgula)	NumI
Sequências de dígitos (com vírgula)	NumR
Sequências de caracteres envolta por aspas	Str

Resposta

Fluxo de Tokens

```
<Delim> <DEC> <Var,"argumento"> <Delim> <INT>
<Var,"fatorial"> <Delim> <INT> <Delim> <ALG> <LER>
<Var,"argumento"> <ATR> <Var,"argumento"> <A>
<Var,"fatorial"> <SE> <Var,"argumento"> <OpRel,"=">
<NumI,"0"> <ENT> <ATR> <NumI,"1"> <A>
<Var,"fatorial"> <ENQ> <Var,"argumento"> <OpRel,">">
<NumI,"1"> <INI> <ATR> <Var,"fatorial"> <OpArit,"*">
<AP> <Var,"argumento"> <OpArit,"-"> <NumI,"1"> <FP>
<A> <Var,"fatorial"> <ATR> <Var,"argumento">
<OpArit,"-"> <NumI,"1"> <A> <Var,"argumento"> <FIM>
<IMP> <Var,"fatorial">
```

Tabela de símbolos

- Nomes de variáveis, funções, classes, etc...
  - Seguem o mesmo padrão
- Portanto é comum tratá-los da mesma forma no analisador léxico
  - Usaremos um token único para todos eles
  - Chamado (quase sempre) de:
    - Identificador
  - O padrão é (quase sempre):
    - Cadeia de letras + números + alguns caracteres (_,$, etc)
    - Mas sempre começando com uma letra

Tabela de símbolos

- É justamente nos casos dos identificadores que entra a necessidade de ciência do contexto
  - Que torna as linguagens de programação não livres de contexto (como vimos na aula passada)
- Para esses casos, ao invés de armazenar o lexema no próprio token
  - Ex: x = 10 ➔ <id,”x”>
- Criamos uma estrutura separada (tabela de símbolos)
  - E no token, inserimos um ponteiro para essa estrutura
  - Ex: x = 0 ➔ <id,312>

311	...
312	x
313	...

Tabela de símbolos

- Toda vez que um mesmo identificador aparece
  - Reaproveitamos a mesma entrada na tabela
- Ex:

int x = z; ➔ <id,312> <id,313>

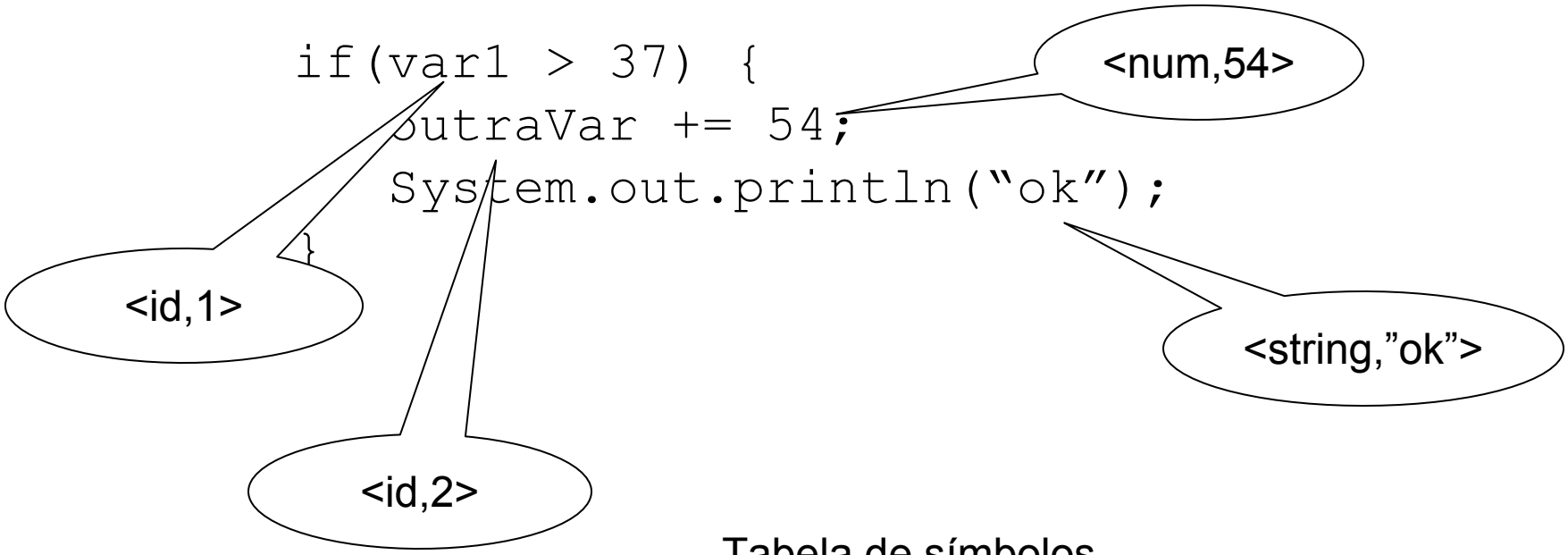
if(x>a) ... ➔ <id,312> <id,314>

Na verdade, depende do escopo (veremos depois)

311	...
312	x
313	z
314	a
315	...

Lexema vs padrão vs token

- Alguns tokens reconhecidos neste exemplo:



Exercício

Identifique os tokens e construa a tabela de símbolos, usando os padrões da linguagem ALGUMA

```
:DECLARACOES
numero1:INT
numero2:INT
numero3:INT
aux:INT
:ALGORITMO
% Coloca 3 números em ordem crescente
LER numero1
LER numero2
LER numero3
SE numero1 > numero2 ENTAO
   INICIO
      ATRIBUIR numero2 A aux
      ATRIBUIR numero1 A numero2
      ATRIBUIR aux A numero1
   FIM
```

Padrão	Sigla
DECLARAÇÕES, ALGORITMO, INTEIRO, REAL, ATRIBUIR, A, LER, IMPRIMIR, SE, ENTAO, ENQUANTO, INICIO, FIM	3 primeiras letras
*, /, +, -	OpArit
<, <=, >=, >, =, <>	OpRel
E, OU	OpBool
:	Delim
(,)	AP / FP
Seq. de letras ou números que começam com letra minúscula	Var
Sequências de dígitos (sem vírgula)	NumI
Sequências de dígitos (com vírgula)	NumR
Sequências de caracteres envolta por aspas	Str

Resposta

Fluxo de Tokens

```
<Delim> <DEC> <Var,1> <Delim> <INT>
<Var,2> <Delim> <INT> <Var,3> <Delim>
<INT> <Var,4> <Delim> <INT> <Delim>
<ALG> <LER> <Var,1> <LER> <Var,2>
<LER> <Var,3> <SE> <Var,1> <OpRel,”>”>
<Var,2> <ENT> <INI> <ATR> <Var,2> <A>
<Var,4> <ATR> <Var,1> <A> <Var,2>
<ATR> <Var,4> <A> <Var,1> <FIM>
```

Tabela de Símbolos

Entrada	Lexema
1	numero1
2	numero2
3	numero3
4	aux
...	

Tabela de símbolos

- Pode ter mais informações
  - Por exemplo: tipo de variáveis, tipo de retorno de funções, etc...
  - As quais serão utilizadas mais adiante no processo de compilação

Lexema vs padrão vs token

- Classes típicas de tokens
  1. Um token para cada palavra-chave (obs: o padrão é o próprio lexema ou a própria palavra-chave)
  2. Tokens para os operadores
     - Individualmente ou em classes (como aritméticos vs relacionais vs booleanos)
  3. Um token representando todos os identificadores
  4. Um ou mais tokens representando constantes, como números e cadeias literais
  5. Tokens para cada símbolo de pontuação
     - Como parênteses, dois pontos, etc...

Erros léxicos

- Erros simples podem ser detectados
  - Ex: 123x&$33
- Mas para muitos erros o analisador léxico não consegue sozinho
- Por exemplo:

```
while (i>3) { }
```

- É um erro léxico, mas como saber sem antes analisar a sintaxe?
- A culpa é dos identificadores
  - O padrão dos identificadores é muito abrangente, então quase tudo se encaixa
- Em outras palavras, quase sempre tem um padrão que reconhece o lexema

Como implementar?

Implementação

- Agora que compreendemos O QUE o analisador léxico deve fazer
  - Vamos começar a estudar COMO ele irá fazê-lo
- Ou seja, aspectos de implementação

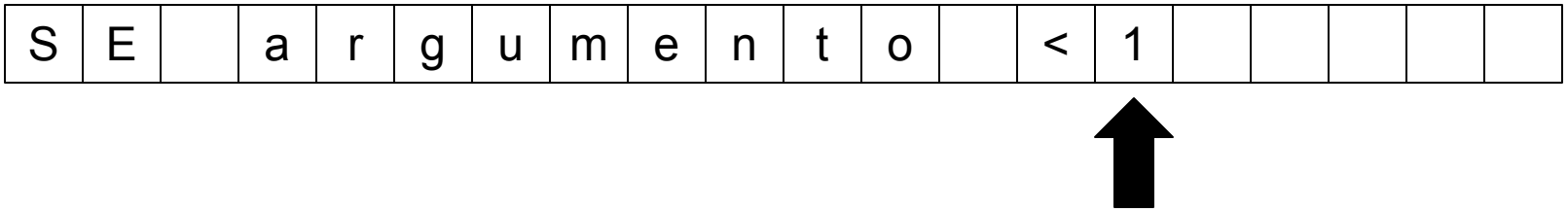
Para “sentir o drama”

- Vamos tentar implementar um analisador léxico para a linguagem ALGUMA

Demonstração

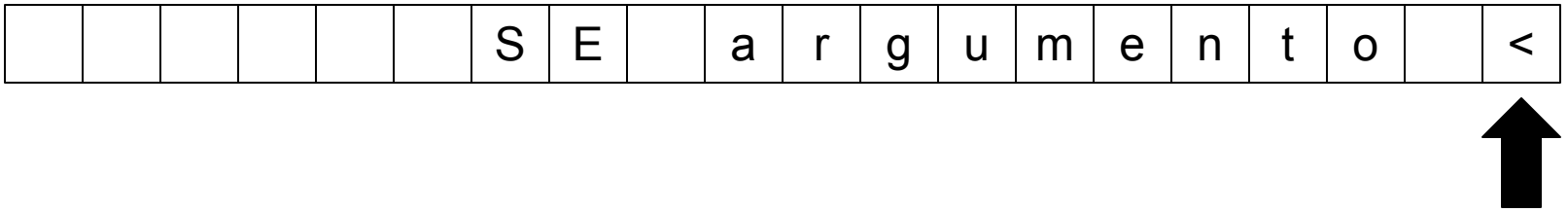
Problema

- Ler a entrada caractere por caractere é ruim
  - Em algumas situações, é preciso retroceder
- Além disso, é ineficiente
- Solução: usar um buffer
  - Um ponteiro aponta para o caractere atual
  - Sempre que chegar ao fim, recarregamos o buffer
  - Caso necessário, basta retroceder

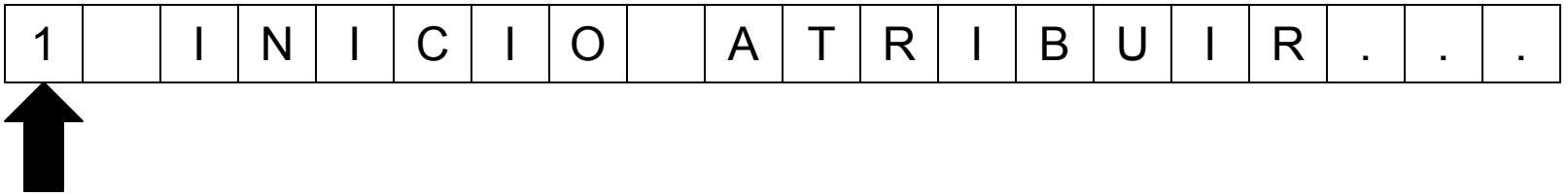


Buffer

- O buffer único tem um problema
  - Veja o exemplo (extremo) abaixo



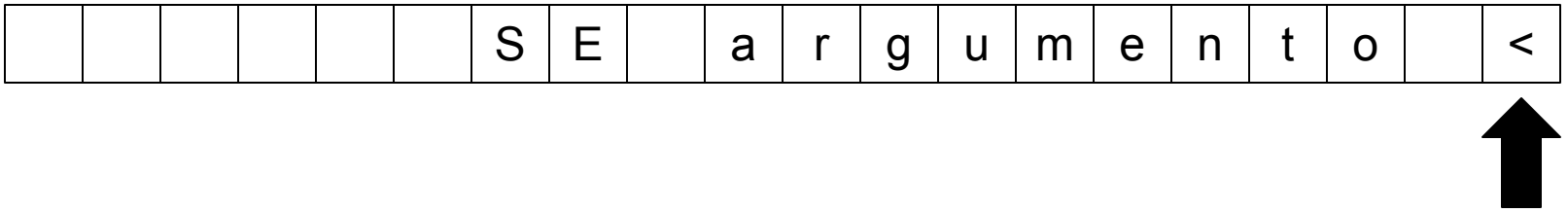
Chegou no fim... recarregando...



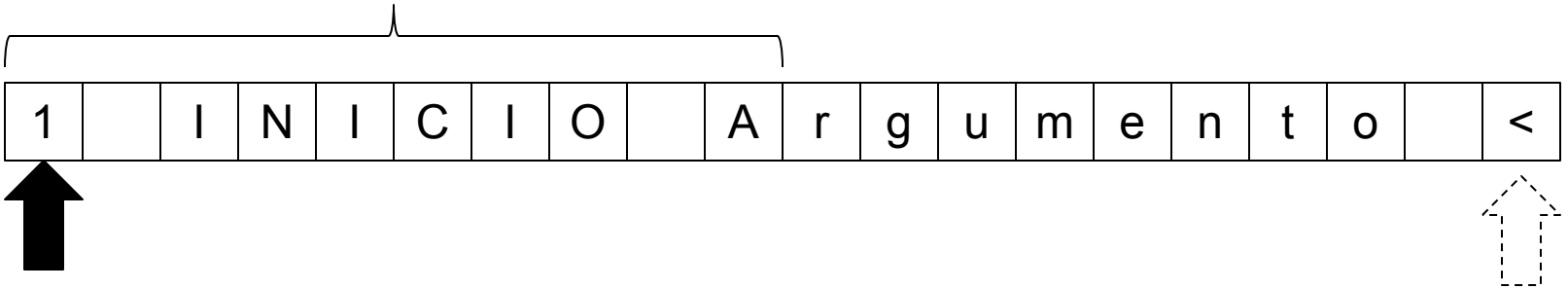
Para retroceder, é necessário recarregar o buffer anterior!

Buffer duplo

- Portanto, é comum o uso de um buffer duplo



Chegou no fim... Recarrega somente a “metade”...



Se precisar retroceder, é só voltar à outra “metade”

Implementação

- Vamos implementar o buffer duplo

Demonstração

Continuando

- Como traduzir os padrões em código?

Padrão	**Sigla**
DECLARAÇÕES, ALGORITMO, INTEIRO, REAL, ATRIBUIR, LER, IMPRIMIR, SE, ENTAO, ENQUANTO, INICIO, FIM	3 primeiras letras
*, /, +, -	OpArit
<, <=, >=, >, =, <>	OpRel
E, OU	OpBool
:	Delim
(,)	AP / FP
Seq. de letras ou números que começam com letra minúscula	Var
Sequências de dígitos (sem vírgula)	NumI
Sequências de dígitos (com vírgula)	NumR
Sequências de caracteres envolta por aspas	Str

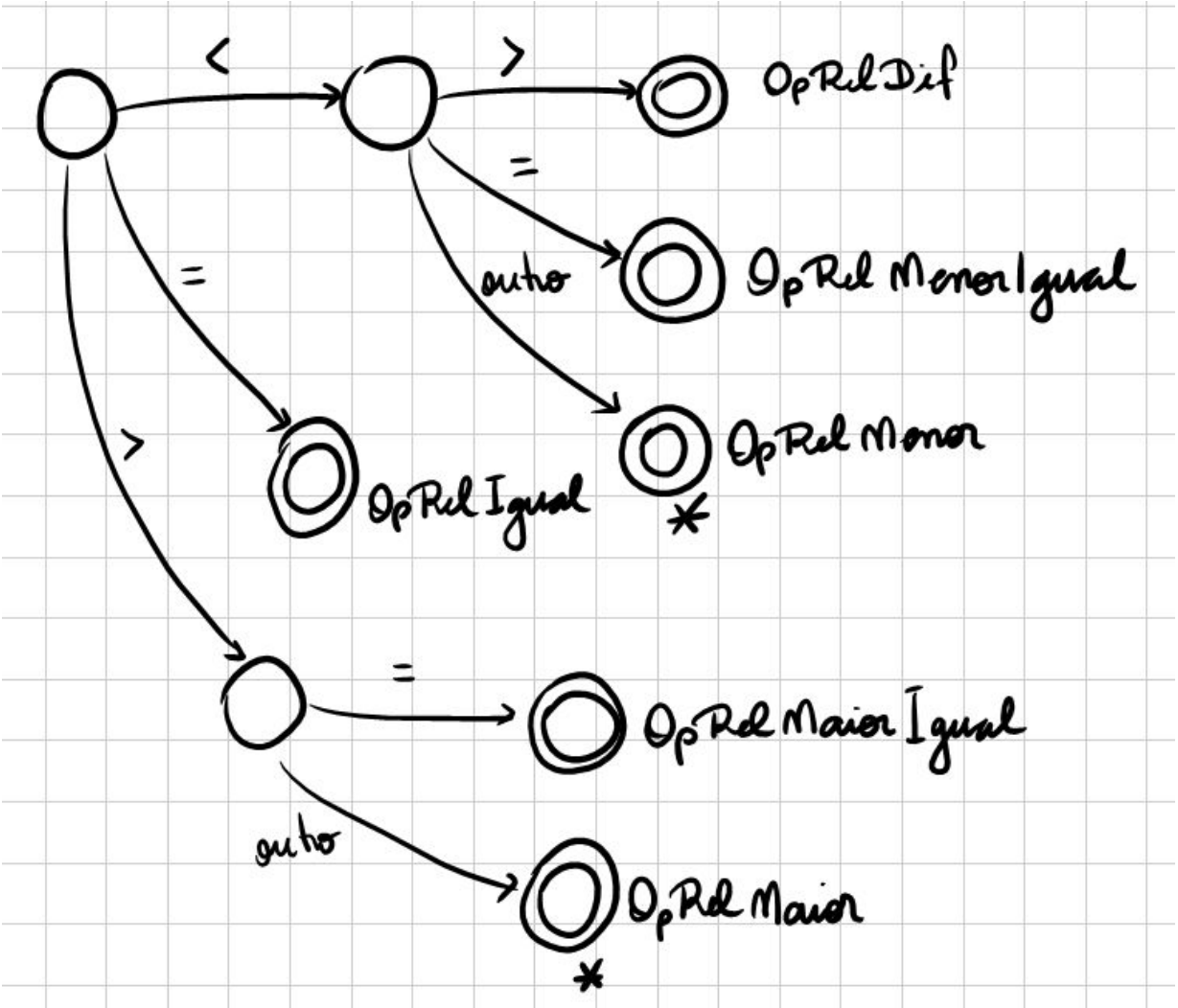
Reconhecimento de padrões

- Opção 1
  - Observações
    - Obs1: para aqueles alunos que não fizeram (ou foram mal) em LFA
    - Obs2: como era feito antigamente
  - Basta implementar a lógica utilizando nossa criatividade
    - E ferramentas disponíveis na maioria das LP

Diagramas de transição

- Diagrama de estados
  - Modelo visual que facilita a implementação da lógica do reconhecimento
- Autômato finito determinístico
  - Até quem foi mal em LFA conhece um AFD:
    - Conjunto finito de estados
      - Estado inicial
      - Estados de aceitação (mais de um)
    - Transições (caracteres)
- Uma extensão
  - Para indicar se é necessário retroceder

Diagramas de transição



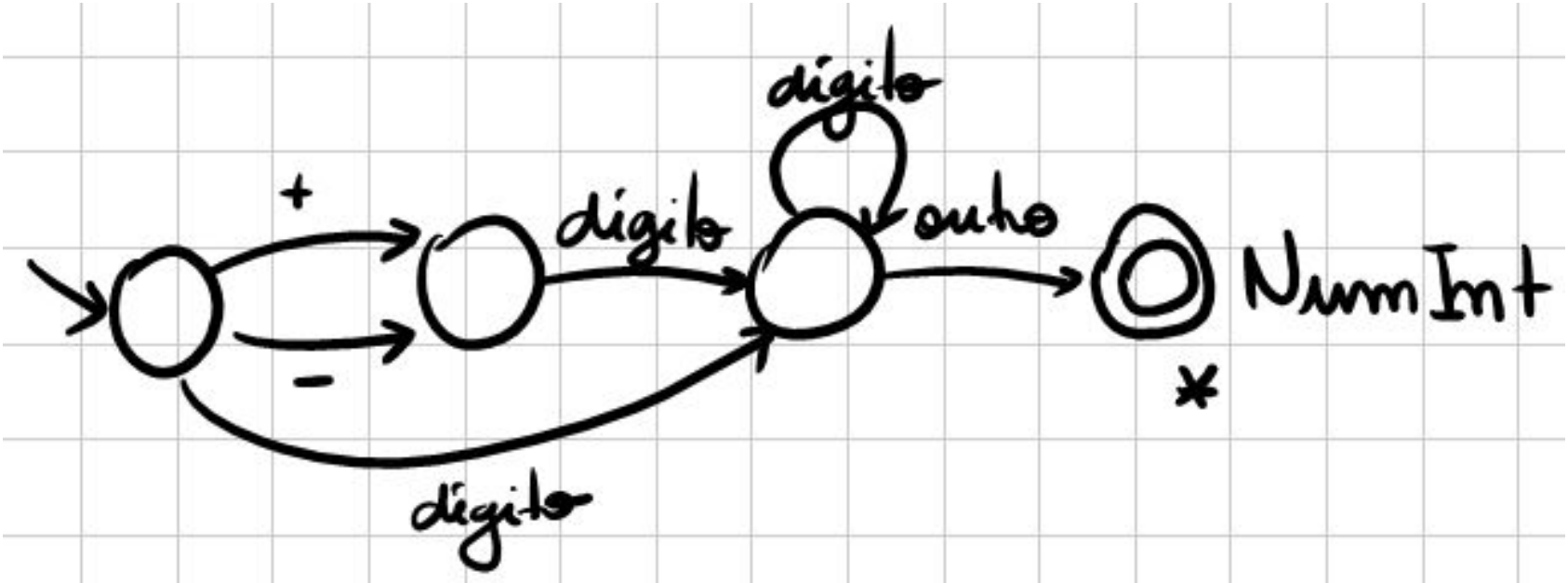
Exercício

- Crie AFDs para reconhecer:
  - Números inteiros (NumInt)
    - +1000, -2, 00231, 2441
  - Números reais (NumReal)
    - +1000.0, -0.50, 0.314
    - Obs: .50, +.22, 30. NÃO são válidos

Exercício

- Crie AFDs para reconhecer:
  - Números inteiros (NumInt)
    - +1000, -2, 00231, 2441

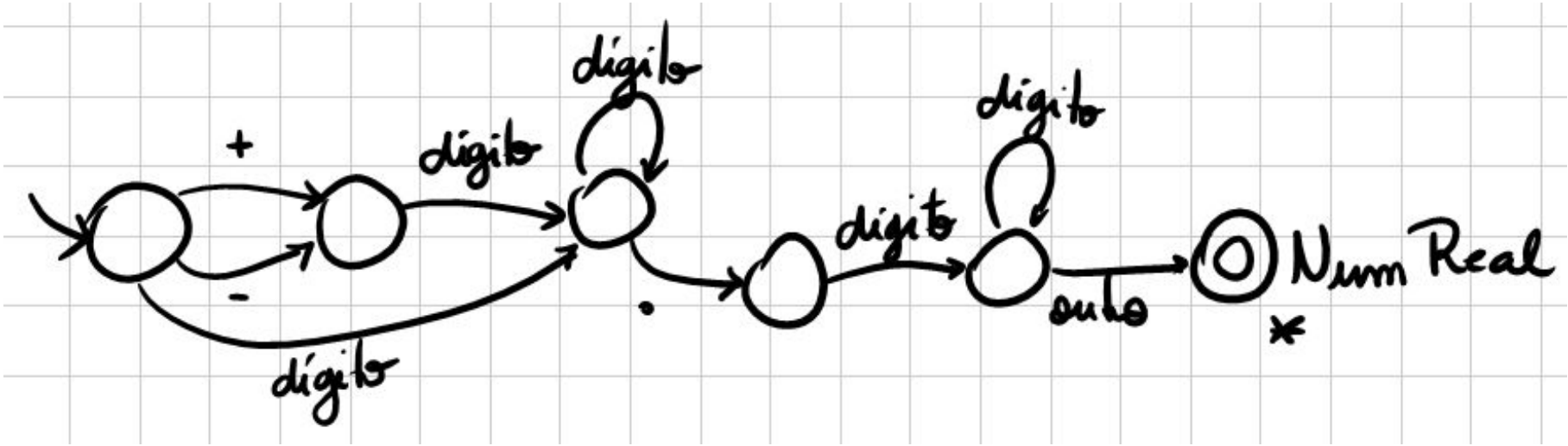
Resposta



Exercício

- Crie AFDs para reconhecer:
  - Números reais (NumReal)
    - +1000.0, -0.50, 0.314
    - Obs: .50, +.22, 30. NÃO são válidos

Resposta



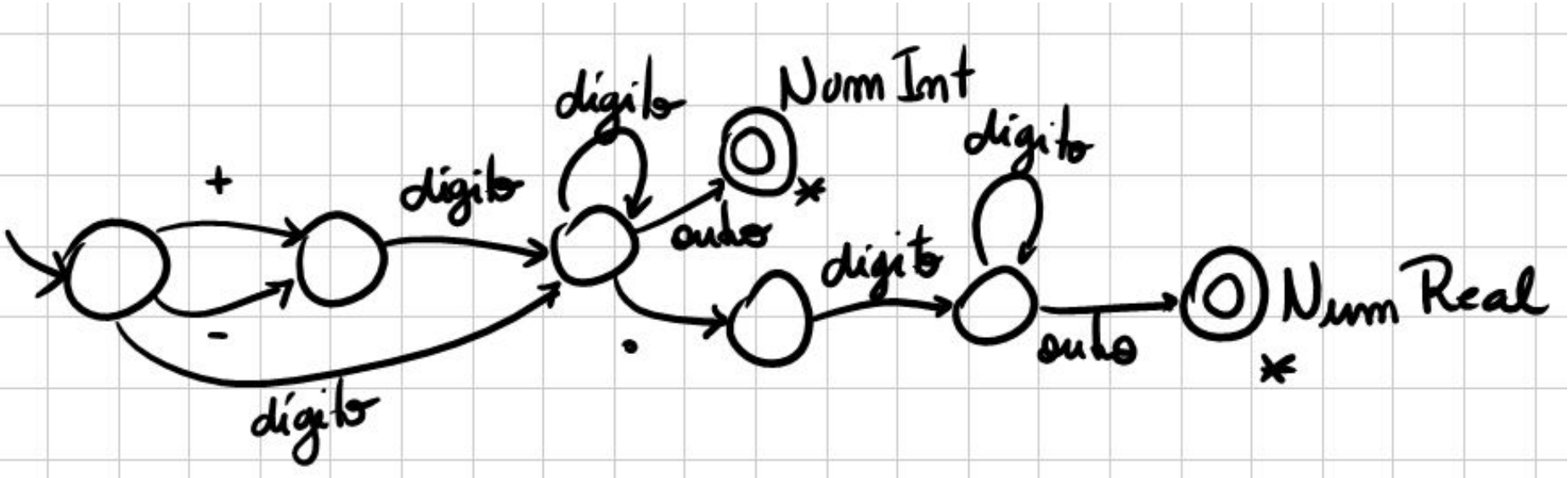
Exercício

- Junte ambos os diagramas em um só

Exercício

- Junte ambos os diagramas em um só

Resposta



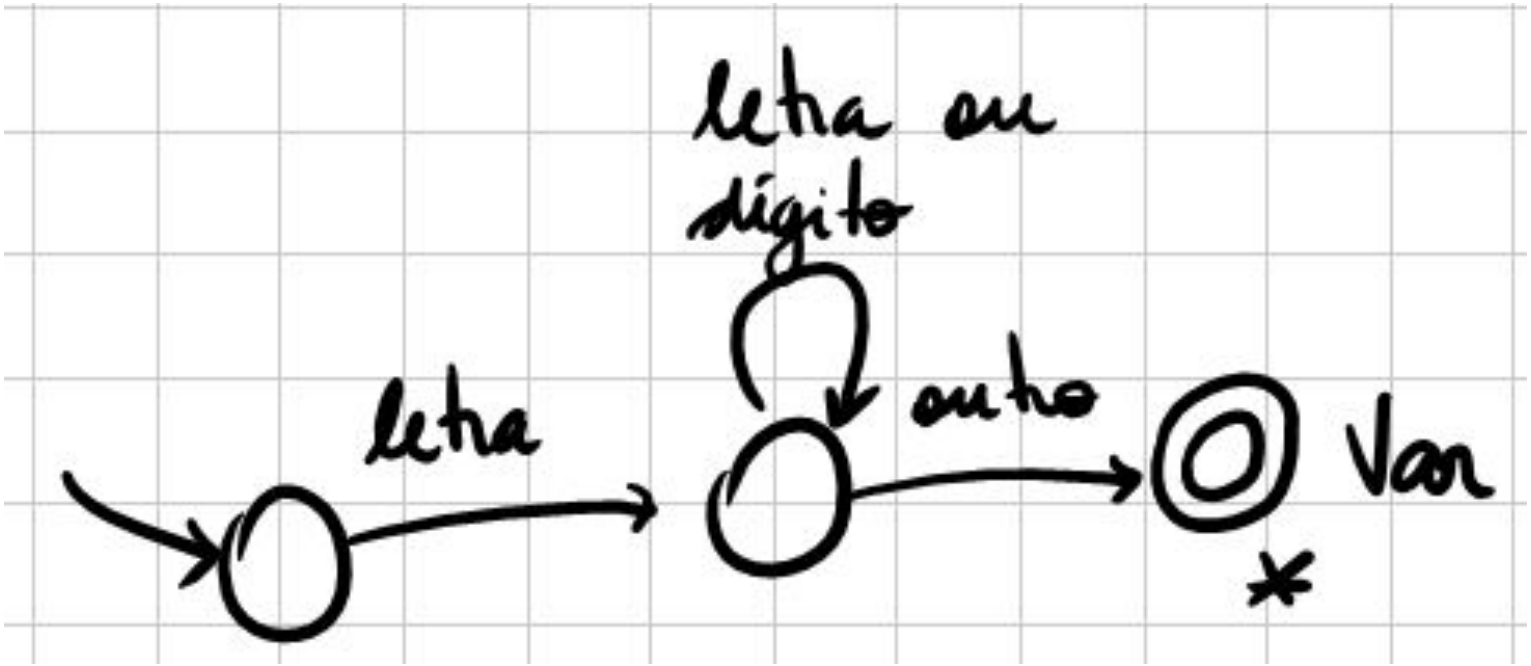
Exercício

- Crie um AFD para reconhecer
  - Identificadores (Var, na linguagem ALGUMA)
    - Sequência qualquer de letras e números, sendo que o primeiro caractere deve ser uma letra
    - Ex: teste, var1, numVoltas52
    - Obs: 123abc, _teste, OutraVar$ NÃO são válidos

Exercício

- Crie um AFD para reconhecer
  - Identificadores (Var, na linguagem ALGUMA)

Resposta



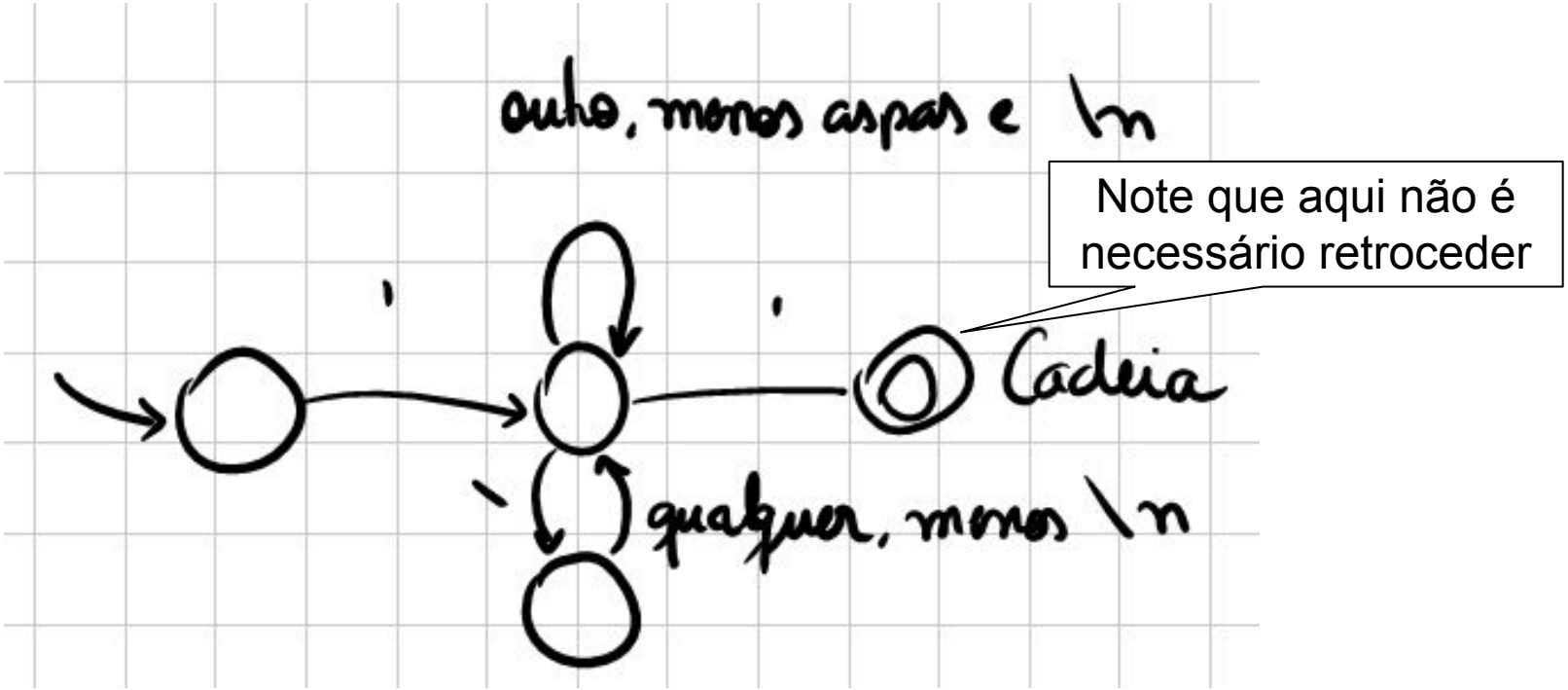
Exercício

- Crie um AFD para reconhecer
  - Cadeia: sequência de caracteres delimitados por aspas simples
    - Ex: ‘Testando’, ‘Uma cadeia qualquer’, ‘#$$A$ASD’
    - Obs: para permitir que a aspas simples seja usada dentro de uma cadeia, utilize o caractere de escape
      - Ex: ‘Cadeia que usa \’ aspas \’ simples dentro’
    - Obs: não pode haver quebra de linha (\n) dentro de uma cadeia

Exercício

- Crie um AFD para reconhecer
  - Cadeia: sequência de caracteres delimitados por aspas simples

Resposta



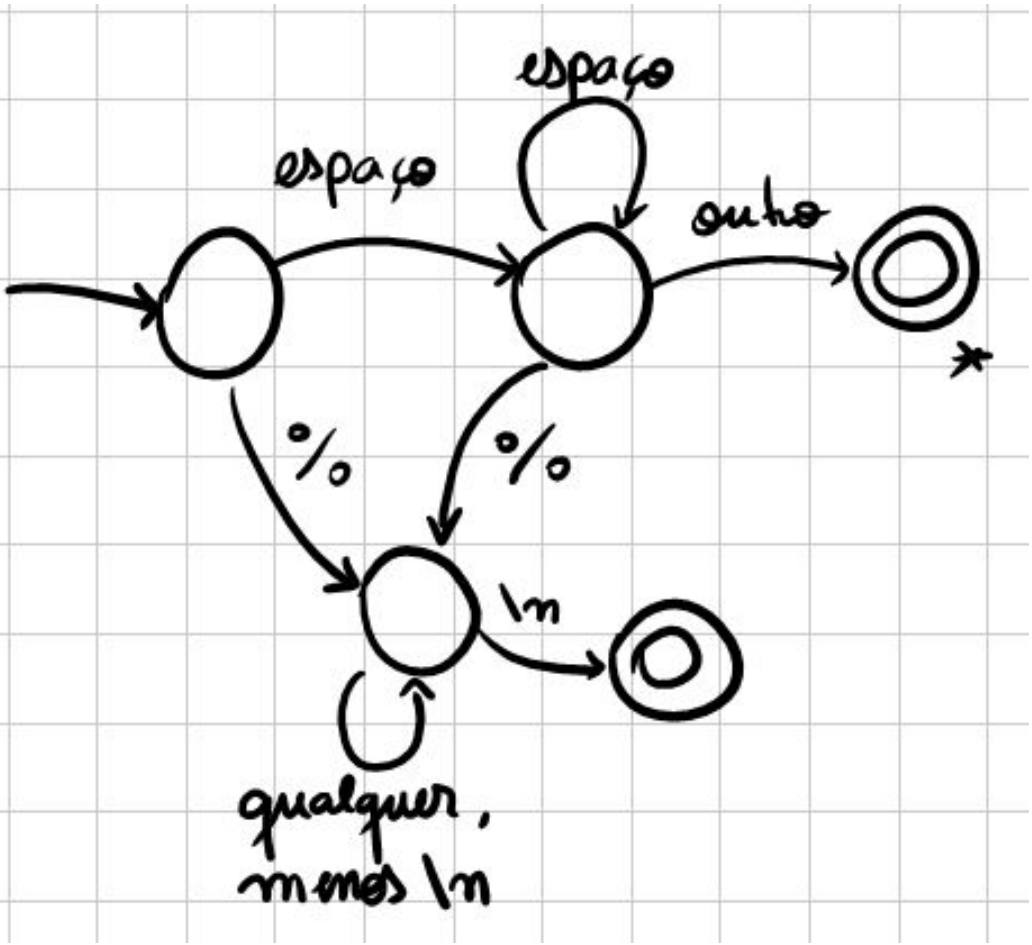
Exercício

- Crie um AFD para reconhecer
  - Comentários (tudo entre % e \n)
  - Espaços em branco (incluindo \n, \t, etc)
  - Obs: faça em um único diagrama

Exercício

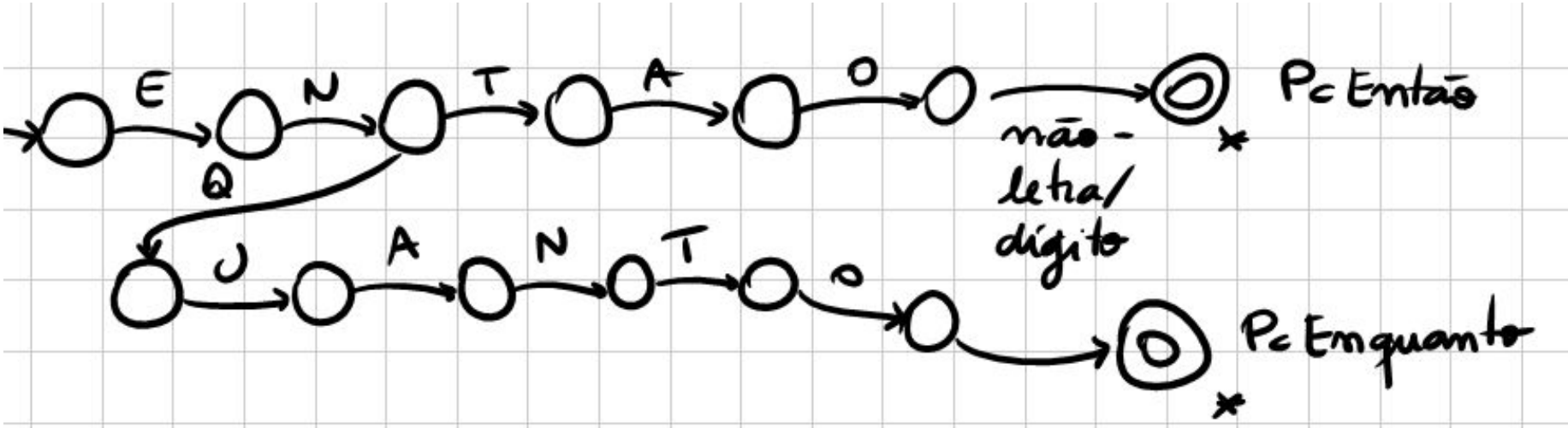
- Crie um AFD para reconhecer
  - Comentários (tudo entre % e \n)
  - Espaços em branco (incluindo \n, \t, etc)
  - Obs: faça em um único diagrama

Resposta



Palavras-chave

- Palavras-chave são simples
- Pode-se combinar os prefixos para simplificar



Implementação

- Queremos um método para “zerar” o lexema atual
  - E outro para “confirmar”

- Depois implementamos cada padrão em um método
- E definimos uma cascata de regras:
  - Ordem lógica (Ex: identificadores depois de palavras-chave)

- O método *proximoToken* testa regra a regra na ordem
- Para cada padrão:
  - Se for bem sucedido, “confirmar”
  - Caso não seja, “zerar”

Demonstração

Implementação

- Agora veremos como é uma tabela de símbolos na prática:
  - Como criá-la
  - Como preenchê-la
  - Como utilizá-la para reconhecer palavras-chave de um modo mais simples

Demonstração

Implementação

- Problemas com essa implementação:
  - Trabalhosa
    - Criação e manutenção
    - Imaginem regras mais complicadas
  - Não eficiente
    - Abordagem tentativa-e-erro
    - Muito vai-e-vém na entrada

Outra opção

- Testar todas as regras “simultaneamente”

- A cada leitura de caractere, testar todas as regras
- E fazer a correspondência com a cadeia mais longa reconhecida
  - Corresponde a mesclar os diagramas anteriores
  - Isso melhora a questão da eficiência

- Mas ainda tem problema na manutenção do código

Opção preferida

- Usando o que aprendemos em LFA, podemos:
  - Especificar padrões usando Expressões Regulares
  - Converter automaticamente
    - Expressões Regulares → AFN → AFD
  - Minimizar estados para aumentar eficiência

Opção preferida

- Especificamos:
  - Letra → [a-zA-Z] { código }
  - Digito → [0-9] { código }
  - Variavel → Letra(Letra|Digito)* { código }

- Um sistema automático gera uma implementação
  - Criação instantânea
  - Manutenção facilitada

Demonstração

- Utilizando:
  - ANTLR (ANother Tool for Language Recognition)
    - *“powerful parser generator for reading, processing, executing, or translating structured text or binary files.”*
    - *“It's widely used to build languages, tools, and frameworks.”*
    - *“From a grammar, ANTLR generates a parser that can build and walk parse trees.*”
  - http://www.antlr.org/
    - Quickstart

Geradores de analisadores léxicos

Teoria e conceitos

Especificação de tokens

- Na prática, expressões regulares são a melhor opção
- Não dá para representar tudo
  - Mas é suficiente para análise léxica
- Possibilita implementação **automática**

Relembrando conceitos

- Alfabeto:
  - Conjunto de símbolos possíveis
    - {0,1}, {a,b}, unicode, ASCII
- Cadeia/palavra/string:
  - Sequência finita de símbolos do alfabeto
- Linguagem:
  - Conjunto de cadeias sobre um alfabeto

Expressões regulares

- Descrevem linguagens regulares

- Linguagem regular:
  - Tipo especial de linguagem
  - Associado aos autômatos finitos determinísticos/não-determinísticos
  - Formada pela aplicação de **operações regulares** sobre outras linguagens regulares

Operações regulares

- Concatenação
  - LM = {st | s está em L e t está em M}
- União
  - L U M = {s | s está em L ou s está em M}
- Fecho
  - L^* = todas as combinações de palavras de L
- Fecho positivo
  - $L^+ = L^* - \{ \varepsilon \}$

Exercícios

- Sejam:
  - L = {A,B,C,...,Z,a,b,...,z}
  - D = {0,1,2,...,9}

- Defina:
  - L U D
  - LD
  - L*
  - L(LUD)*
  - D^+

Respostas

- L U D = {A,B,C,...,Z,a,b,c,...,z,0,1,2,...,9}
- LD = 520 cadeias de tamanho 2
  - {A0,A1,A2,...,A9,B0,B1,...,B9,...,z0,z1,...,z9}
- L* = Conjunto de todas as cadeias de letras
- L(LUD)* = Conjunto de todas as cadeias de letras e dígitos começando com uma letra
- D^+ = Conjunto de todas as cadeias de um ou mais dígitos

Expressões regulares

- Notação para descrever linguagens regulares

- Símbolos que representam as operações regulares
  - (r)|(s) = L(r) U L(s) (união)
  - (r)(s) = L(r)L(s) (concatenação)
  - (r)* = (L(r))* (fecho/estrela Kleene)
  - $(r)^+$ = (L(r))+ (fecho positivo)
  - (r) = L(r) (parênteses servem para agrupar)

Exercícios

- Descrever os padrões em palavras
  - a|b → {a,b}
  - (a|b)(a|b) → {aa,ab,ba,bb}
  - a|a*b → {a,b,ab,aab,aaab,aaaab,...}
  - a(a|b)+ → {aa,ab,aabba,aaaab,aaaaa,...}

Definições regulares

- Regras auxiliares
  - Conveniência notacional
  - Consiste em dar nomes a algumas expressões comuns e usá-los em outras expressões
  - **Não pode haver recursividade!**

- Exemplos:
  - Letra → a | b | c ... | z
  - Digito → 0 | 1 | 2 | ... | 9
  - Id → Letra (Letra | Digito)*

Expressões regulares no ANTLR

- Usaremos a notação do ANTLR como referência
- As regras seguem o formato:

```
Nome : definição ;
```

- Nas regras léxicas, o nome geralmente começa com letra maiúscula

Expressões regulares no ANTLR

- Símbolos do alfabeto aparecem entre aspas simples
  - Caso seja necessário especificar aspas simples, utilize \'
  - Para aspas duplas não precisa (“” é válido)

- Exemplos:

```
Teste : 'a' | 'b' | 'c';
Entao: 'entao';
TipoVar: 'inteiro' | 'real' |
  'booleano';
Outra: 'testando \' com aspas';
Outra2: 'testando " com aspas'
```

Expressões regulares no ANTLR

- Intervalos podem ser utilizados

- Exemplos:

```
LetraMinuscula : ('a'..'z');
Digito : '0'..'9';
```

Expressões regulares no ANTLR

- Símbolos:
  - * (fecho)
  - + (fecho positivo)
  - ? (opcional)
  - ~ (negação)
  - . (qualquer caractere)

- Exemplos:

```
Numero : ('+'|'-')?('0'..'9')+;
Id: Letra (Letra|Digito|'_')*;
Cadeia : '"' (~('\\'|'"') )* '"';
Tudo : .+;
```

Expressões regulares no ANTLR

- Palavra-chave *fragment* denota definições regulares auxiliares

- Exemplos:

```
fragment
LetraMinuscula : ('a'..'z');

fragment
Digito : '0'..'9';

Id: LetraMinuscula (Digito)+;
```

Ambiguidade

- ANTLR permite ambiguidade

- Quando mais de uma expressão corresponde à entrada:
  - A maior sequência é escolhida
  - Se duas ou mais regras correspondem a um mesmo número de caracteres de entrada
    - a regra que aparece primeiro é escolhida

Ambiguidade

- Exemplos:

```
Int : 'integer';     /* regra 1 */
Id  : ('a'..'z')+;   /* regra 2 */
```

- Se entrada == “integers”
  - Ambas aceitam a entrada
  - A regra 2 é escolhida, pois engloba os 8 caracteres

- Se entrada == “integer”
  - Ambas aceitam a entrada
  - A regra 1 é escolhida, pois ambas englobam os 7 caracteres, mas a regra 1 aparece primeiro

Ambiguidade

- Na verdade, o ANTLR nem deixa você declarar o contrário

- Ele acusa um erro na tentativa de geração do seguinte código

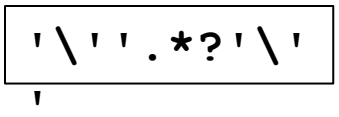
```
Id  : ('a'..'z')+;
Int : 'integer';
```

Ambiguidade

- Esse comportamento “ganancioso” não se reflete nos padrões envolvendo `.*?` (qualquer caractere, zero ou mais vezes seguido de relutante)
  - Exemplo: `'\'' .*? '\''` (sequência de caracteres envoltos em aspas)

- Dada a entrada

```
'primeira' palavra e 'segunda' palavra
```



Sequência reconhecida: ***'primeira'***

Ambiguidade

- ANTLR V4 é ganancioso por padrão

- É necessário cuidado
  - Exemplo: `'\'' ( ~('\n') )* '\''`
    - (sequência de caracteres, com exceção de fim de linha, envoltos em aspas)

- Neste caso o ANTLR será ganancioso e gerará um analisador que “consome” as aspas
  - Ou seja, o analisador não vai funcionar como deveria

Ambiguidade

- Alternativa 1:
  - definir que aquela regra não deve utilizar comportamento ganancioso
    - `'\'' ( ? : ~('\n') )* '\''`

- Alternativa 2:
  - definir explicitamente a condição de parada
  - ou transformar a regra para remover o não-determinismo
    - `'\'' (~('\''|'\n') )* '\''`

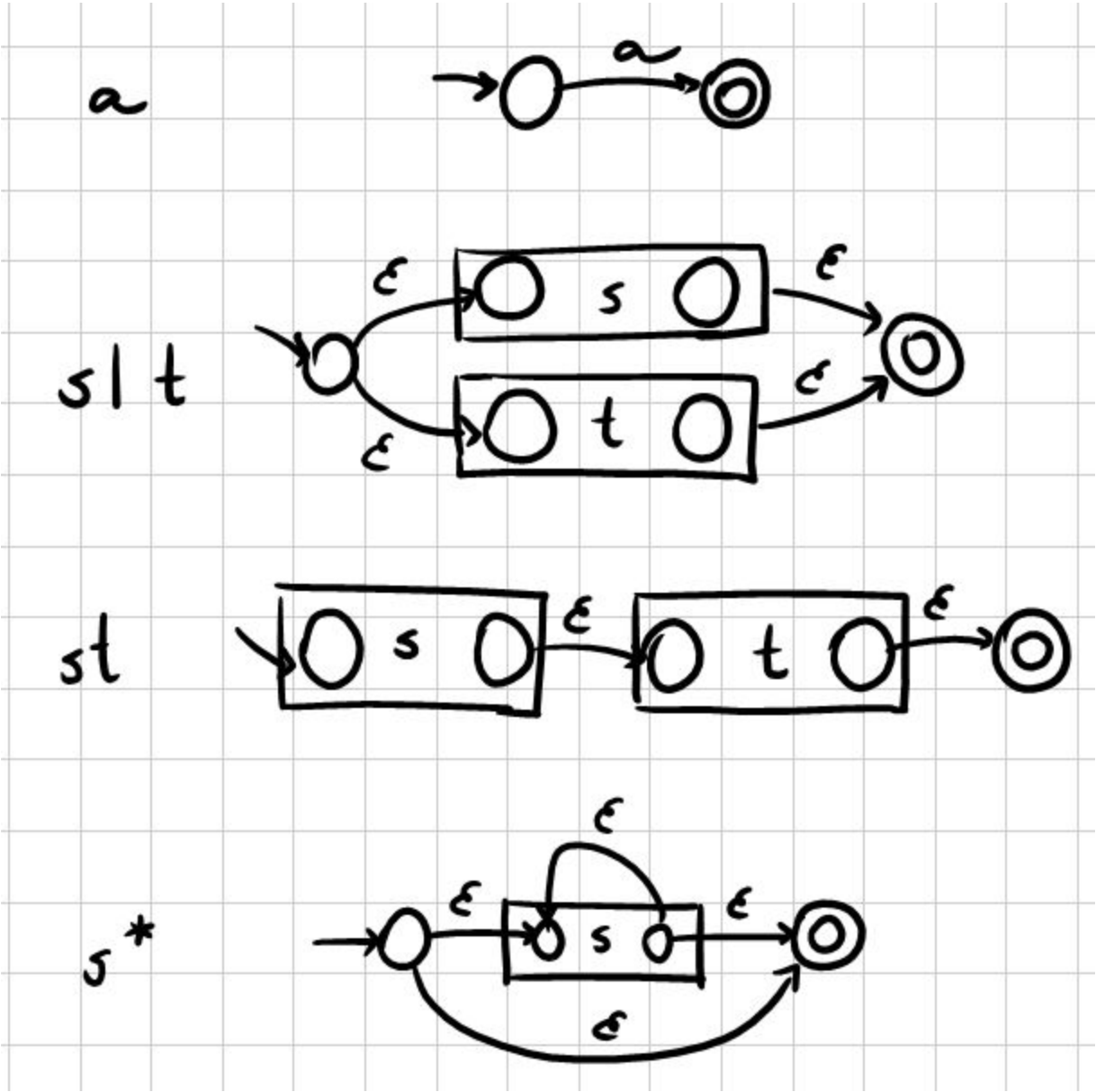
Exercícios

- Vamos praticar na linguagem ALGUMA
- Defina:
  - NumInt
  - NumReal
  - Variavel
  - Cadeia
  - Comentario
  - Espaços em branco (WS)
  - PalavrasChave
  - Operadores e pontuação
- Vamos implementar as respostas diretamente no ANTLR

Demonstração

A ciência por trás

- Expressões regulares → AFN
  - Algoritmo de McNaughton-Yamada-Thompson
  - Recursivamente divide ER, e constrói AFN a partir sub - expressões



A ciência por trás

- Notação estendida para ERs:
  - $E^+ = EE^*$
  - E? = (E | ε)
  - ‘a’..’z’ = ‘a’ | ‘b’ | ‘c’ | ... | ‘z’
  - ~(‘a’..’c’) = ‘d’ | ‘e’ | ... (todos outros símbolos do alfabeto)

A ciência por trás

- AFN → AFD
  - Método de construção dos subconjuntos
  - Cada estado do AFD representa um conjunto de estados do AFN
    - Todos os estados em que é possível se chegar através de transições vazias e com as entradas lidas
  - As transições do AFD examinam todas as possibilidades de não-determinismo do AFN
    - Simulação “em paralelo”
  - No pior caso, o número de estados do AFD é exponencial em relação ao número de estados do AFN
    - Mas isso quase nunca acontece na prática

A ciência por trás

- Ao invés de converter AFN para AFD, é possível executar o AFN diretamente
  - É o mesmo procedimento, mas executado "on-the-fly"
- Porém, isso pode gastar tempo
  - É mais recomendado para quando os padrões mudam constantemente
    - Ex: comandos de busca
  - No nosso caso (compiladores), os padrões dos tokens são fixos
    - Então vale a pena gastar um tempo uma vez só para tornar a compilação (análise léxica, no caso) mais rápida

A ciência por trás

- Existe também um algoritmo que converte uma ER diretamente em um AFD
  - Mais eficiente do que usando um AFN intermediário
  - Pode levar a menos estados
  - Mas é mais complexo
    - Quem tiver interesse em implementar seu próprio gerador de analisadores léxico estude o método no livro do dragão

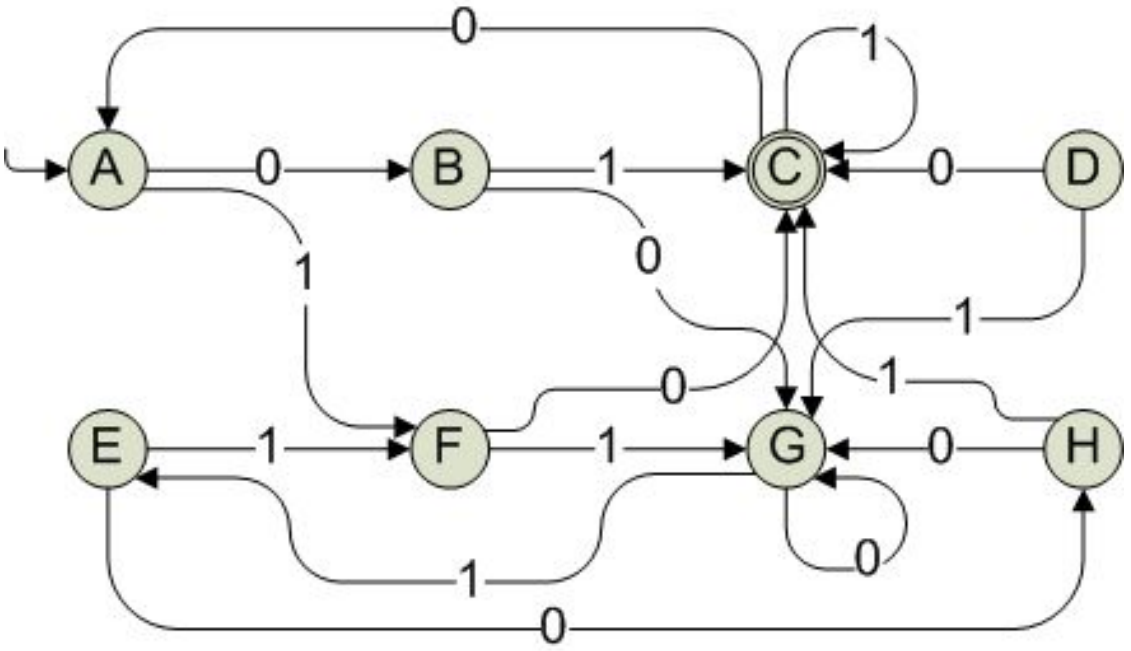
A ciência por trás

- Minimização de estados
  - Sempre existe um AFD com um número mínimo de estados
  - Agrupando estados equivalentes
  - 2 estados são equivalentes se houver uma mesma cadeia que leva cada um a aceitação e não-aceitação, respectivamente
  - O algoritmo particiona estados em grupos de estados que não podem ser distinguidos

A ciência por trás

- Minimização de estados (lembrança de LFA)

	A	B	C	D	E	F	G
B	x						
C	x	x					
D	x	x	x				
E		x	x	x			
F	x	x	x		x		
G	x	x	x	x	x	x	
H	x		x	x	x	x	x



Resultado:
- São pares equivalentes: (A,E), (B,H) e (D,F)

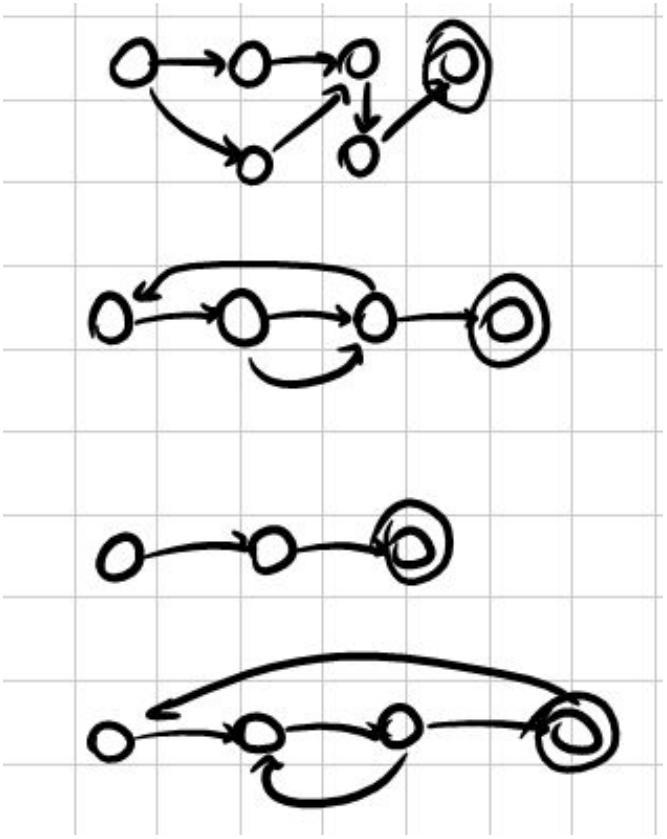
Estratégia para geradores de analisadores léxicos

- 1. Criar uma ER para cada padrão

```
NUMINT : ('+'|'-')?('0'..'9')+;

NUMREAL : ('+'|'-')?('0'..'9')+ ('.' ('0'..'9')+)?;

VARIAVEL : ('a'..'z'|'A'..'Z') ('a'..'z'|'A'..'Z'|'0'..'9')*;

CADEIA : '\'' ( ESC_SEQ | ~('\''|'\\') )* '\'';
```

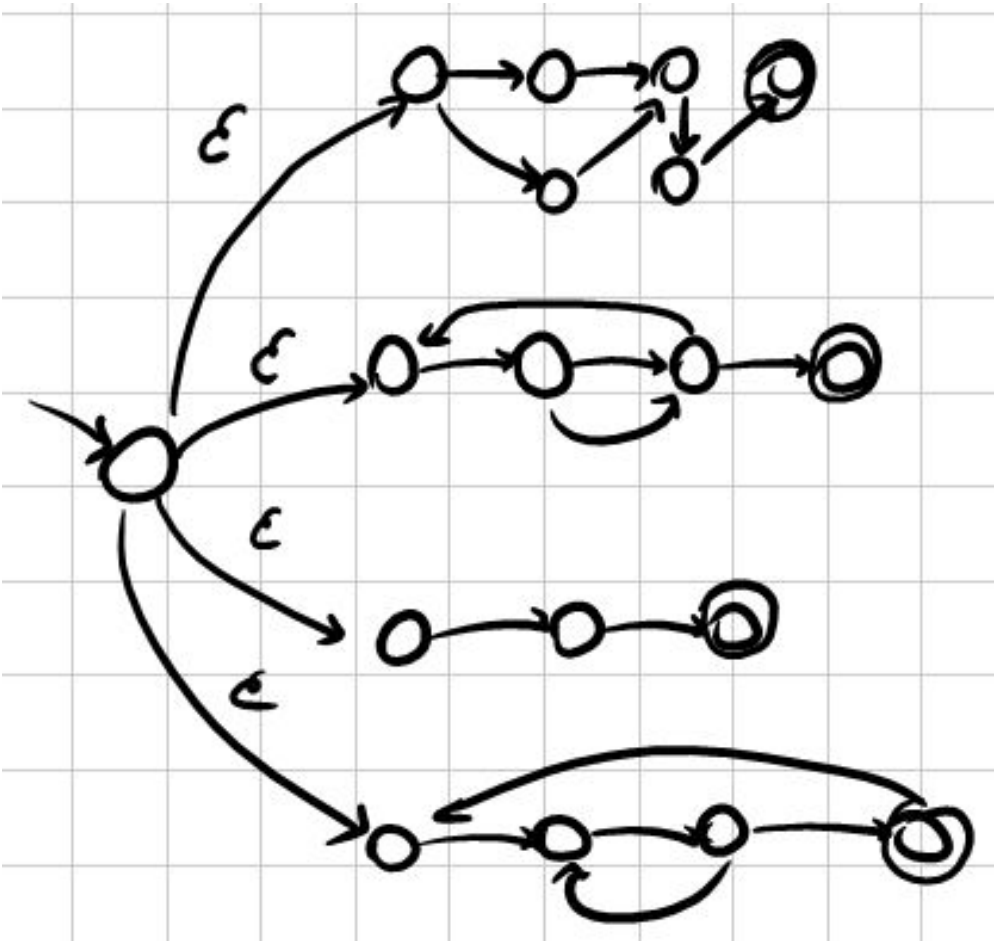
Estratégia para geradores de analisadores léxicos

- 2. Converter cada ER em um AFN
  - Obs: cada estado final corresponde à correspondência de um padrão



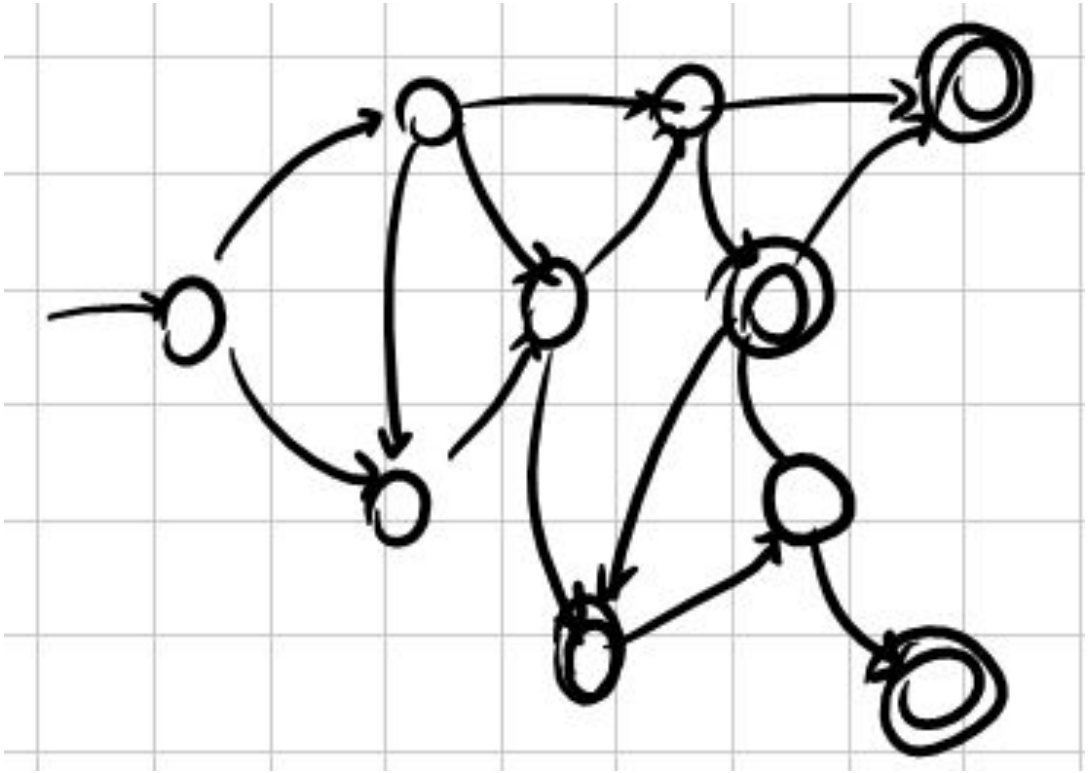
Estratégia para geradores de analisadores léxicos

- 3. Combinar os AFNs individuais em um único



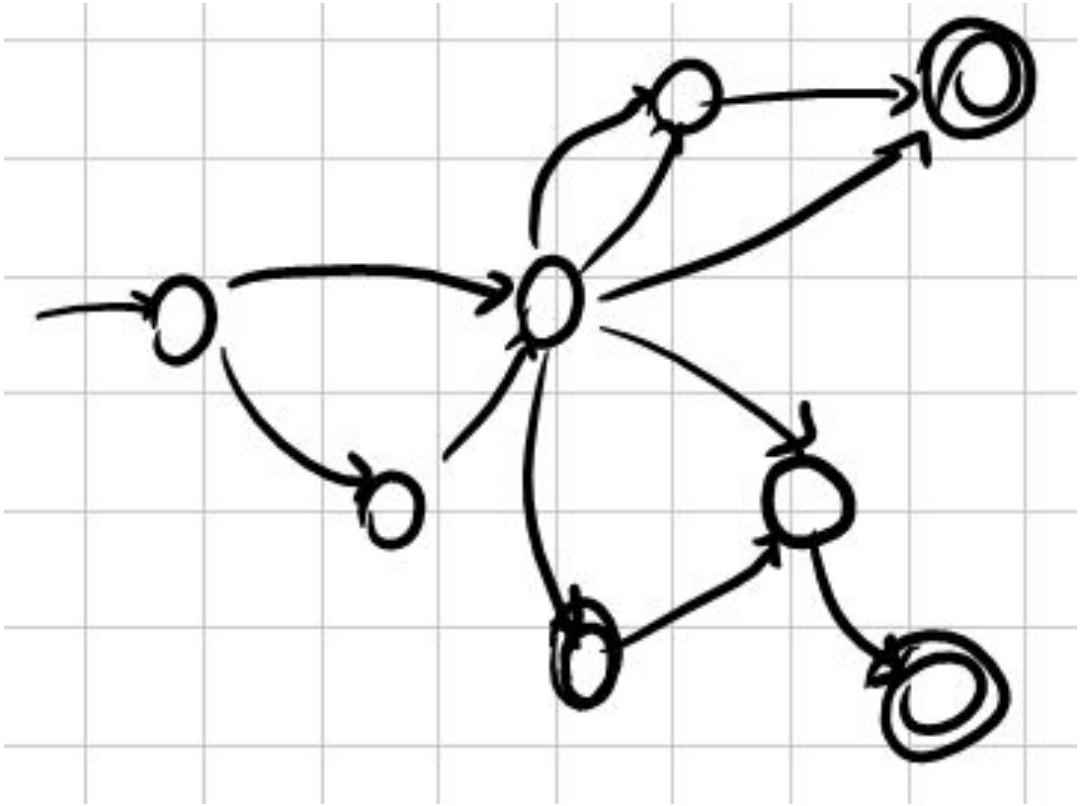
Estratégia para geradores de analisadores léxicos

- 4. Converter o AFN em um AFD



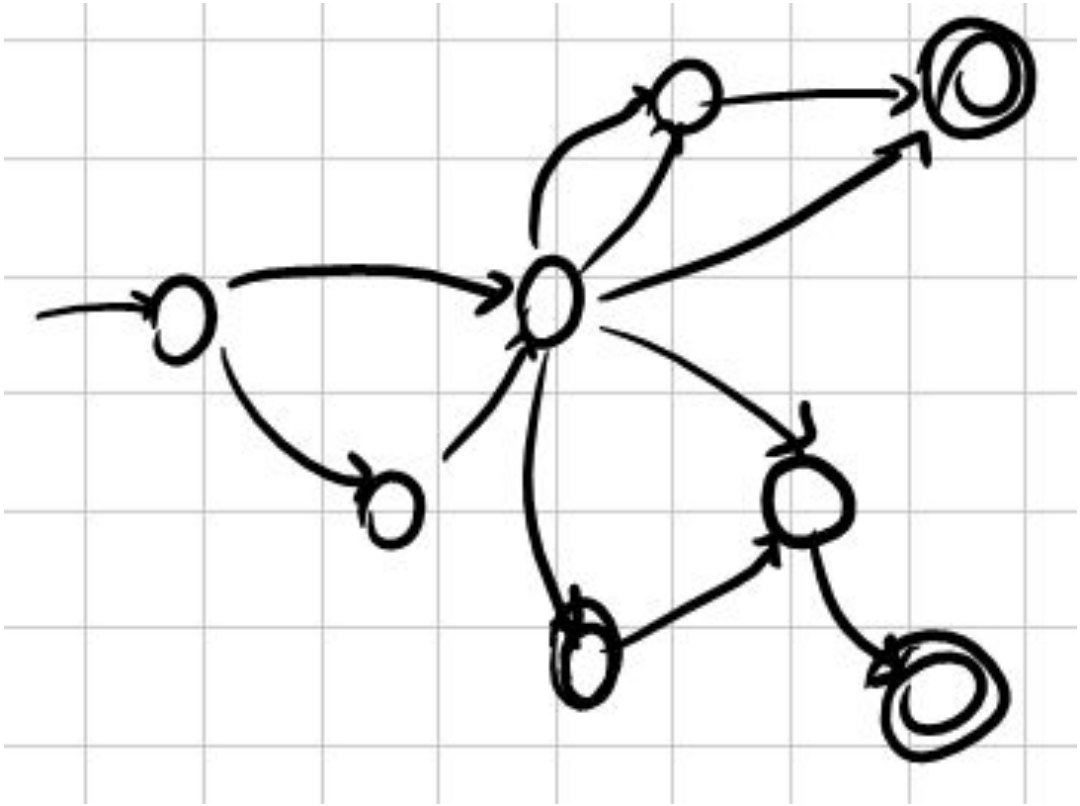
Estratégia para geradores de analisadores léxicos

- 5. Minimizar os estados do AFD



Estratégia para geradores de analisadores léxicos

- 6. Executar o AFD
  - Tabela gerada + algoritmo fixo, por exemplo



Estratégia para geradores de analisadores léxicos

- É possível fazer algumas extensões
  - Decidir por comportamento ganancioso/relutante
  - Operador “lookahead” para ciência de contexto
  - Compactar o tamanho da tabela de transições

Alguns geradores de analisadores léxicos

- Na disciplina usaremos o ANTLR
  - Facilidade de utilização
  - Editor especializado
    - Auto-complete
  - Fácil integração com o analisador sintático
  - Tem suporte a múltiplas linguagens
  - Um dos mais utilizados (segundo o Google)
- Mas existem outros
  - Lex / Flex / Flex++ / Jlex / JFlex
  - Cada um tem uma implementação ligeiramente diferente
  - Mas a ideia geral é a mesma

Para casa

- Examine o código do analisador manual, demonstrado em aula
- Pratique os exemplos da linguagem ALGUMA
  - Analise a definição da linguagem
- Instale e utilize o ANTLR
  - Reproduza os exemplos feitos durante a aula
  - Defina suas próprias expressões
- Para refletir:
  - Como você implementaria um analisador léxico que ignora comentários em múltiplas linhas, começando com /* e terminando com */?
  - Como você poderia implementar um analisador léxico que identifica etiquetas (tags) no formato @texto=valor **dentro de comentários**?

Fim