

Construção de compiladores

Prof. Daniel Lucrédio

Departamento de Computação - UFSCar

1º semestre / 2015

Aula 2

Análise léxica

Contexto

- Alguém assistiu Matrix?
 - (Pergunta besta em um curso de computadores)
- Já pararam pra pensar porque os agentes conversam usando linguagem humana? (começo do 2º filme)
 - Computadores não conversam assim!
 - Na verdade, a linguagem humana é pouco eficiente para computadores conversarem entre si
 - Seria mais parecido com o som do modem conectando
 - (Alguém ainda se lembra?)
- Na verdade, no filme eles também conversam através dos “plugues” no ouvido
 - Modelo mais real
- Apenas nas cenas onde nós (espectadores humanos) precisamos compreender é que eles falam nossa linguagem entre si


Contexto

- Em compiladores:
 - Computador precisa entender um programa
 - Vários motivos (programação, dados, etc)
 - Programação
 - Instruções (cálculos, E/S)
 - Dados
 - Comentários
- É uma **conversa!**
 - É unilateral (ou quase) - homem passando informações para máquina

Contexto

- Sendo uma conversa (mesmo que unilateral)
 - O homem precisa conseguir compreender
 - Para poder formular corretamente, e raciocinar em cima
 - Homem compreende 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
 - Para ele, uma “fala” é só uma sequência de caracteres
 - Que precisa ser compreendida de alguma forma
- As pessoas não pensam muito nisso
 - Parece “óbvio” seguir nosso modelo
 - Mas poderíamos implementar uma máquina (autômato) para reconhecer (e interpretar significado) essa cadeia diretamente
 - Uma máquina de Turing, por exemplo

Contexto

- Seguir o modelo natural é muito útil
- Primeiro motivo: humanos entendem melhor os programas que seguem nosso “modelo” de linguagens
 - Palavras e espaços
- Segundo motivo: facilita a implementação
 - Léxico vs sintático são classes de problemas diferentes
 - Léxico: reconhecer palavras-chave e separadores
 - Sintático: dar estrutura a estas palavras (reconhecer frases)
 - Podemos focar em diferentes partes do problema separadamente (dividir-para-conquistar)

Contexto

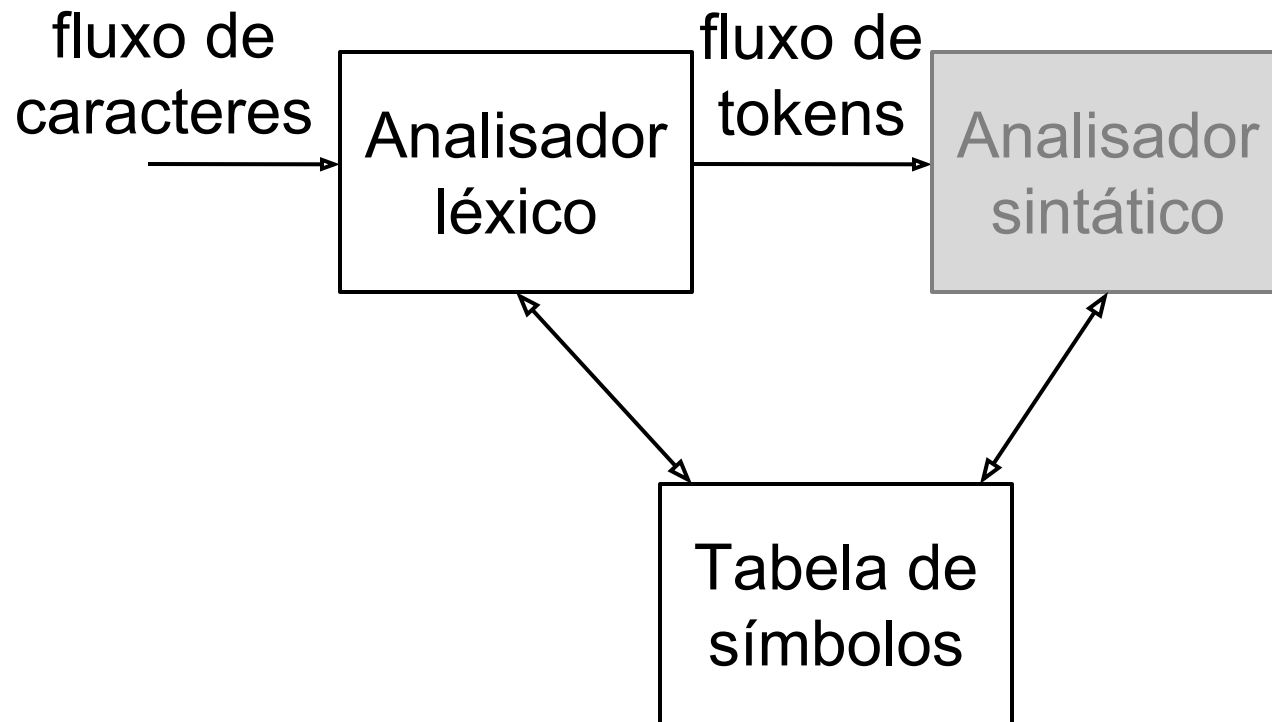
- Terceiro motivo: facilita o trabalho do analisador sintático
 - Opção 1 (tudo junto): na análise sintática, eu teria que lidar com espaços, comentários, nomes de tokens e o problema da sensibilidade ao contexto
 - Opção 2 (separar léxico do sintático):
 - Léxico pode descartar espaços e comentários, e remover nomes, valores e constantes
 - Sintático pode tratar cada CLASSE de lexema como um único símbolo terminal. A gramática fica muito mais simples.

Contexto

- Quarto motivo: eficiência
 - É possível otimizar algumas tarefas de leitura
 - Já que são tarefas especializadas
 - Por exemplo, criar um buffer de entrada no léxico
- Quinto motivo: portabilidade
 - Peculiaridades de leitura não “poluem” o parser
 - Por exemplo, manter o número de linha para melhor reportar erros
 - Isso pode ser tratado no léxico, e portanto o parser fica mais independente (e portátil)

O analisador léxico

Contexto



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

Contexto

- Outras tarefas
 - Remover comentários
 - Remover espaços em branco (tabulações, enter, etc)
 - 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	>	3
)	{	outraVar	+=	5
;	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 categorizá-los de acordo com as suas classes
- 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
 - Estrutura de dados (par tipo/valor) que representa um lexema reconhecido

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

<tipo,valor>

- Tipo do token → usado pelo analisador sintático
- Valor → O próprio lexema ou outras informações
 - Valor numérico, caso o lexema seja uma constante
 - Ponteiro para a tabela de símbolos (veremos depois)

Lexema vs padrão vs token

- Alguns tokens reconhecidos neste exemplo:
 - Conforme os padrões ao lado

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

<num,54>

<if>

<id,"var1">

<cadeia,"ok">

<id,"outraVar">

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	Cad
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
 - Declaração de variáveis (inteiras e reais)
 - Expressões aritméticas (+, -, *, /)
 - Expressões relacionais (>, >=, <=, <, =, <>)
 - 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

Padrão	Tipo de lexema	Sigla
Os próprios lexemas	Palavras-chave: DECLARAÇÕES, ALGORITMO, INTEIRO, REAL, ATRIBUIR, LER, IMPRIMIR, SE, ENTAO, ENQUANTO, INICIO, FIM	3 primeiras letras
Os próprios lexemas	Operadores aritméticos: *,/,+,-	OpArit
Os próprios lexemas	Operadores relacionais: <,<=,>=,>,<>	OpRel
Os próprios lexemas	Operadores booleanos: E, OU	OpBool
Os próprios lexemas	Delimitador: :	Delim
Os próprios lexemas	Parêntesis: (,)	AP / FP
Sequências de letras e 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

Ex: 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, LER, IMPRIMIR, SE, ENTAO, ENQUANTO, INICIO, FIM	3 primeiras letras
*,/,+,-	OpArit
<,<=,>=,>=,<>	OpRel
E, OU	OpBool
:	Delim
(,)	AP / FP
Seq. de letras e 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: $x = 10 \rightarrow \langle \text{id}, "x" \rangle$
- Criamos uma estrutura separada (tabela de símbolos)
 - E no token, inserimos um ponteiro para essa estrutura
 - Ex: $x = 0 \rightarrow \langle \text{id}, 312 \rangle$

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:

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

<id,1>

<id,2>

<num,54>

<string,"ok">

Tabela de símbolos

Entrada	Lexema
1	var1
2	outraVar

Ex: 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, LER, IMPRIMIR, SE, ENTAO, ENQUANTO, INICIO, FIM	3 primeiras letras
*,/,+,-	OpArit
<,<=,>=,>=,<>	OpRel
E, OU	OpBool
:	Delim
(,)	AP / FP
Seq. de letras e 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...
 - 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

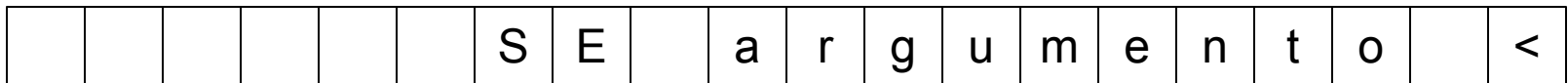
- Ler a entrada caractere por caractere é ruim
 - Em algumas situações, é preciso retroceder
- Além disso, é ineficiente
- Solução: usar um buffer
 - Um ponteiro aponta para o caractere atual
 - Sempre que chegar ao fim, recarregamos o buffer
 - Caso necessário, basta retroceder

S	E		a	r	g	u	m	e	n	t	o		<	1					
---	---	--	---	---	---	---	---	---	---	---	---	--	---	---	--	--	--	--	--

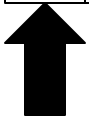
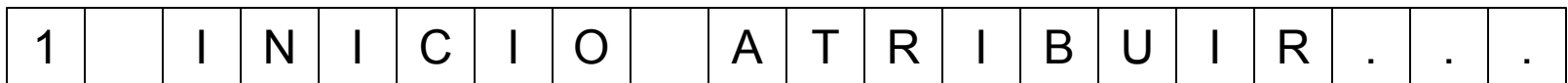


Buffer

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



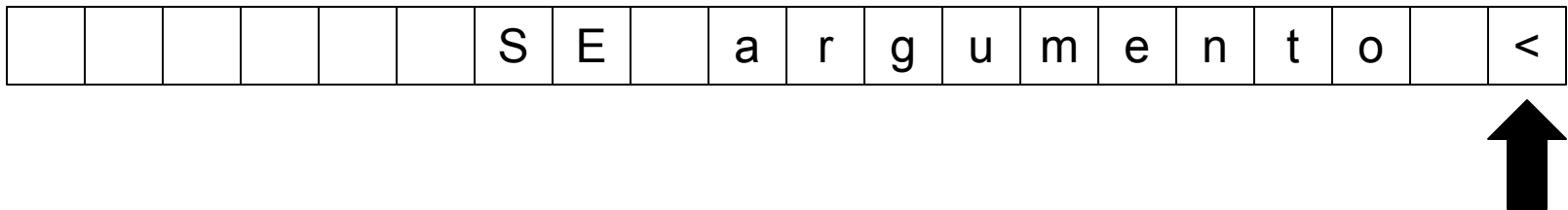
Chegou no fim... recarregando...



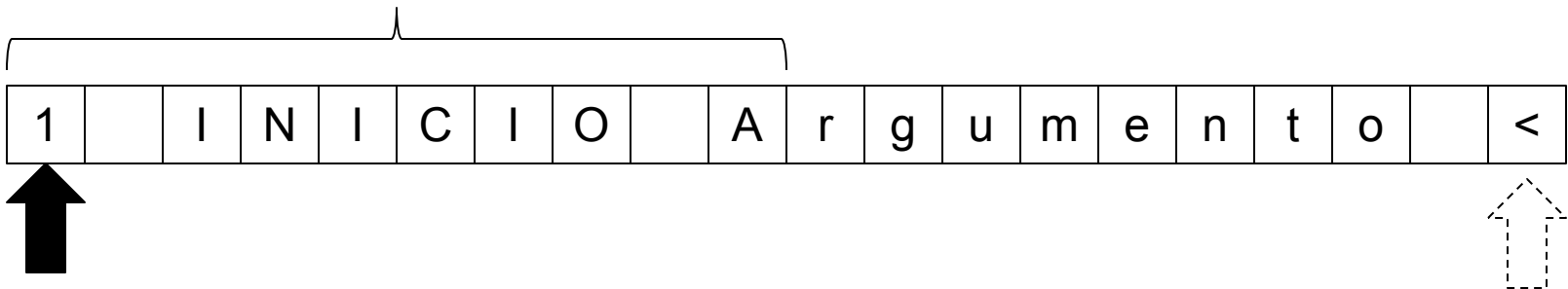
Para retroceder, é necessário recarregar o buffer anterior!

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 e 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

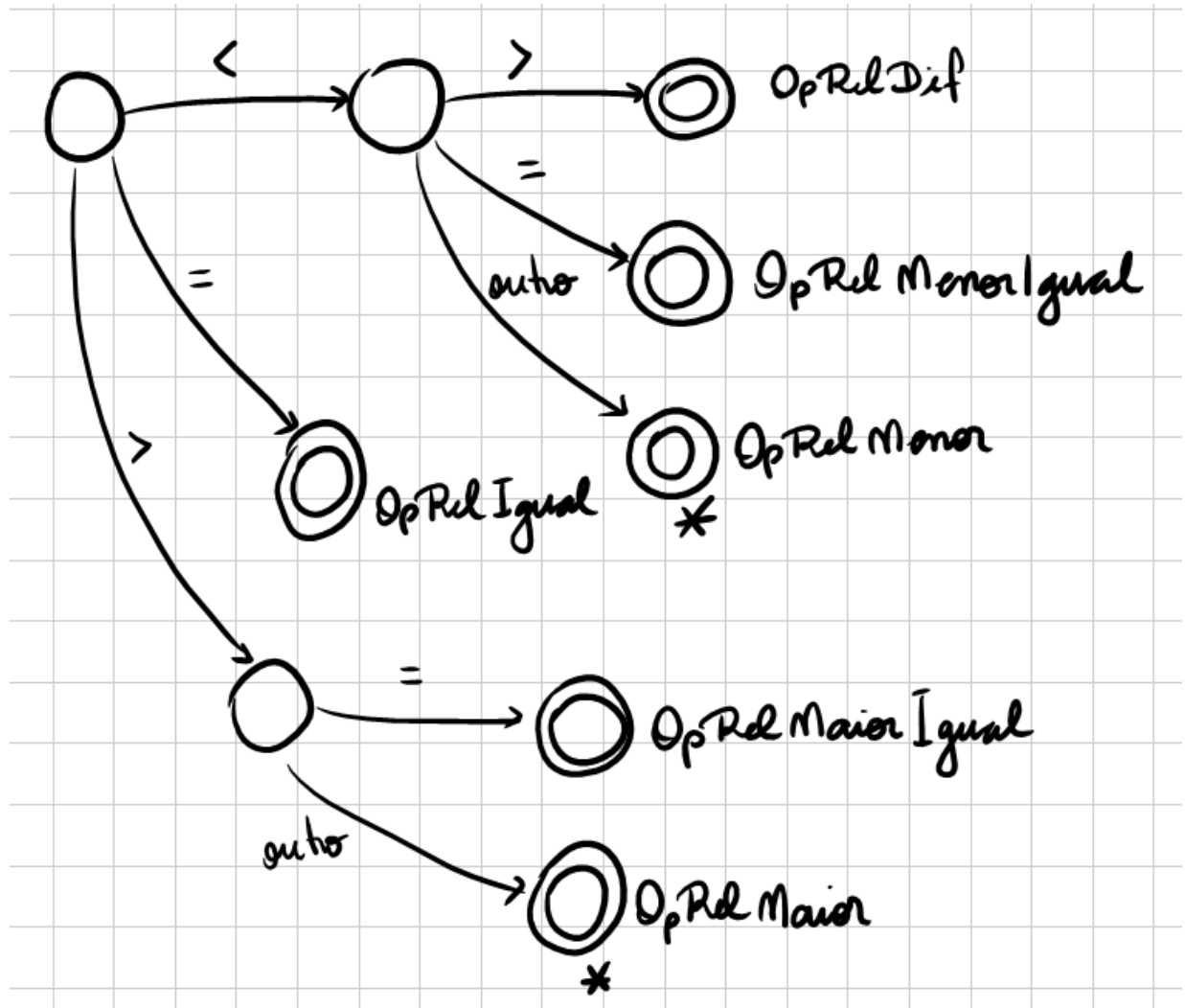
Reconhecimento de padrões

- Opção 1
 - Obs: 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
 - Porque até quem foi mal em LFA conhece um AFD
 - Estado inicial
 - Estados de aceitação (mais de um)
 - Transições (caracteres)
- Uma extensão
 - Para indicar se é necessário retroceder

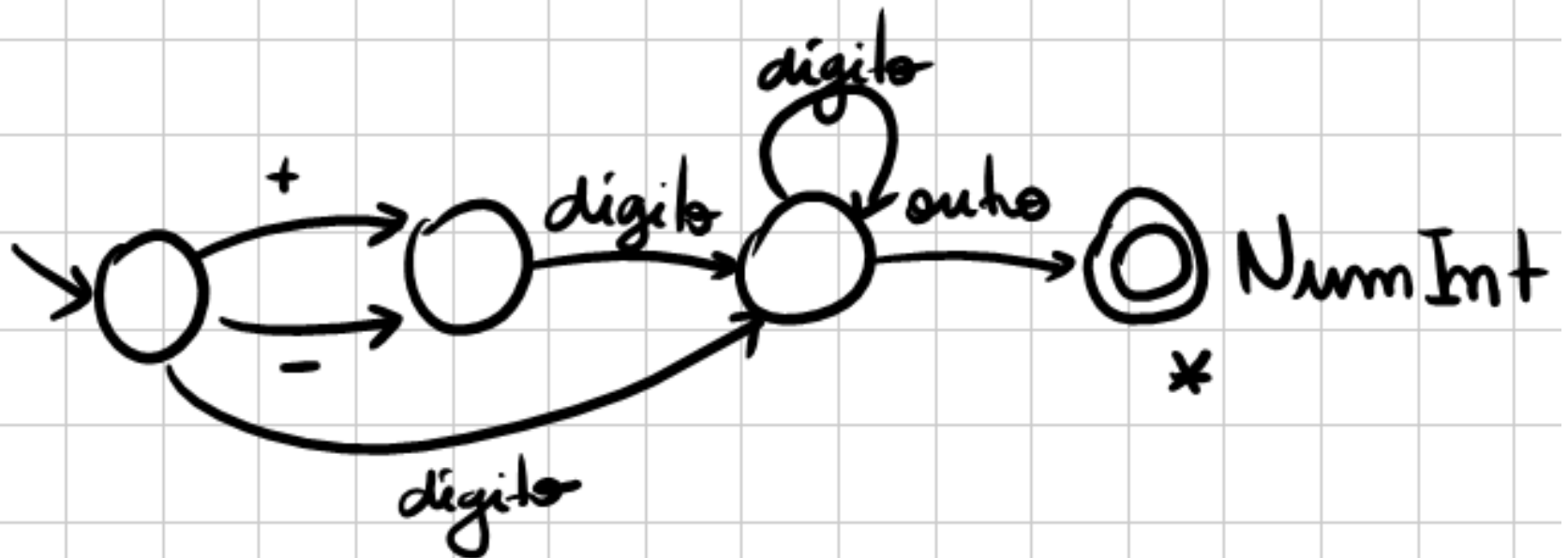
Diagramas de transição



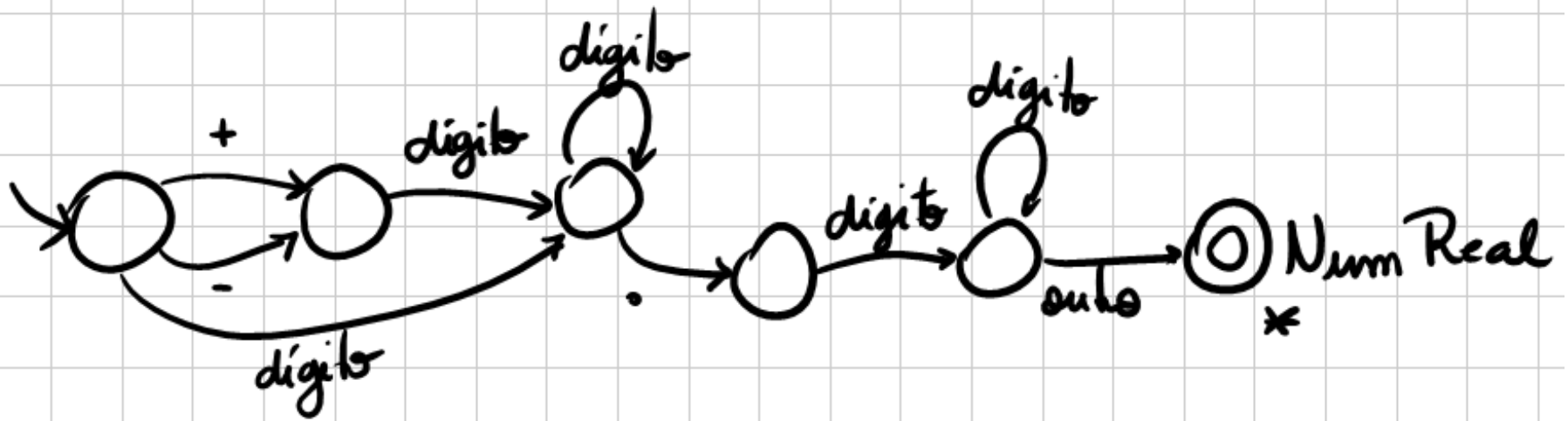
Exercício

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

Respostas

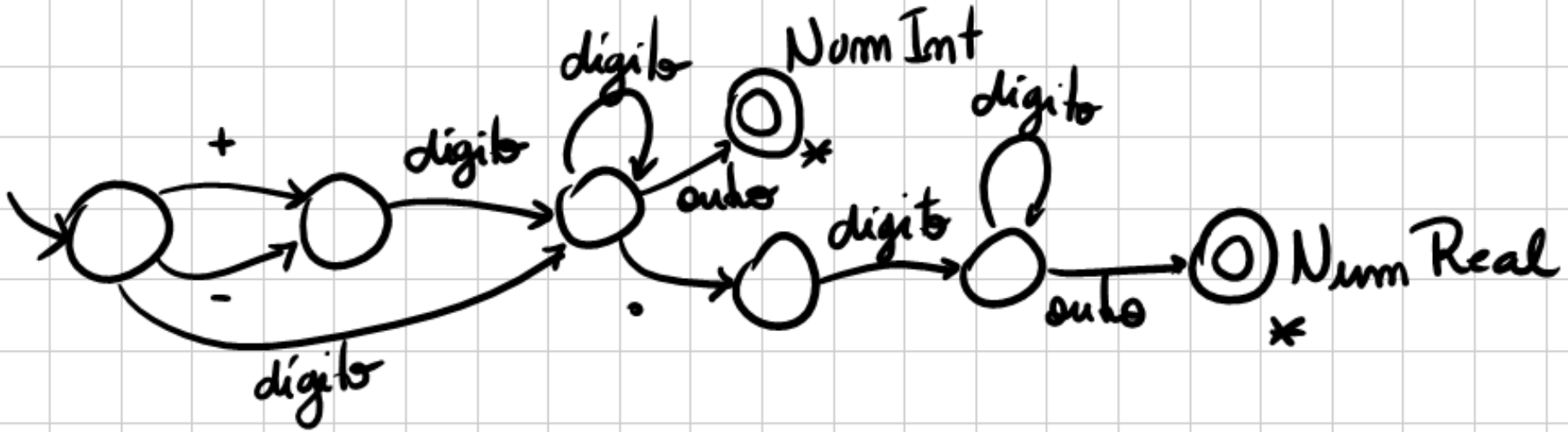


Respostas



Exercício

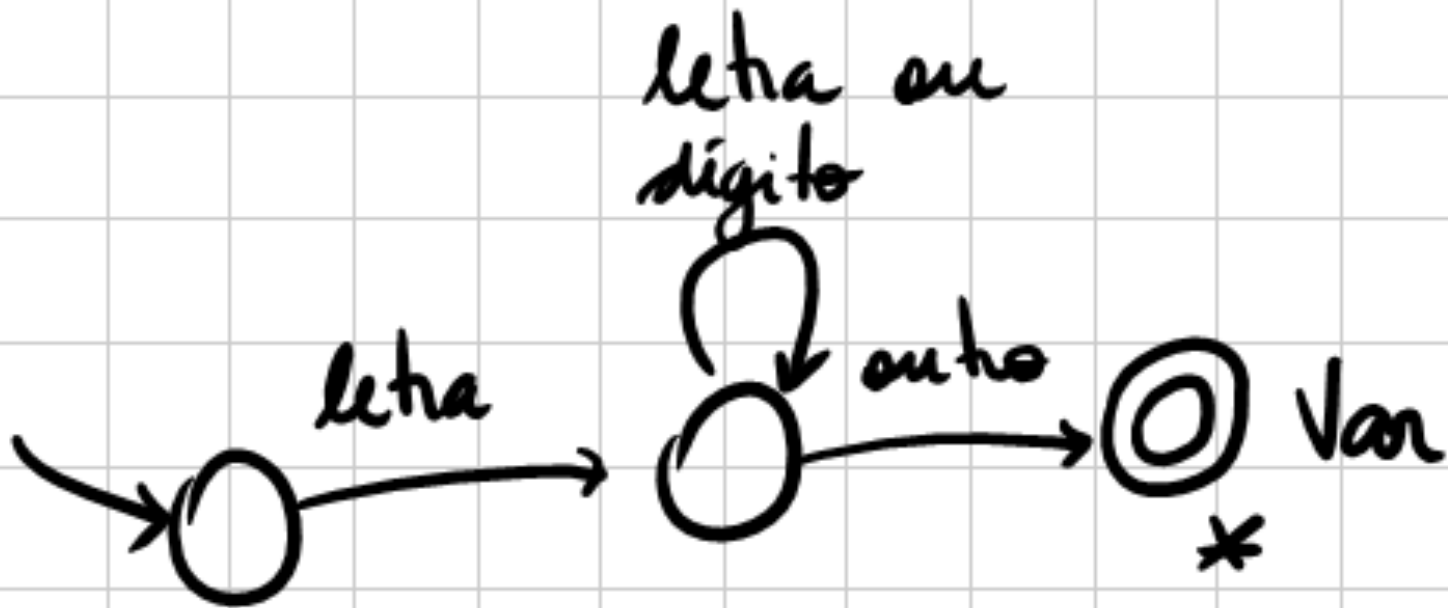
- Junte ambos os diagramas em um só



Exercício

- 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

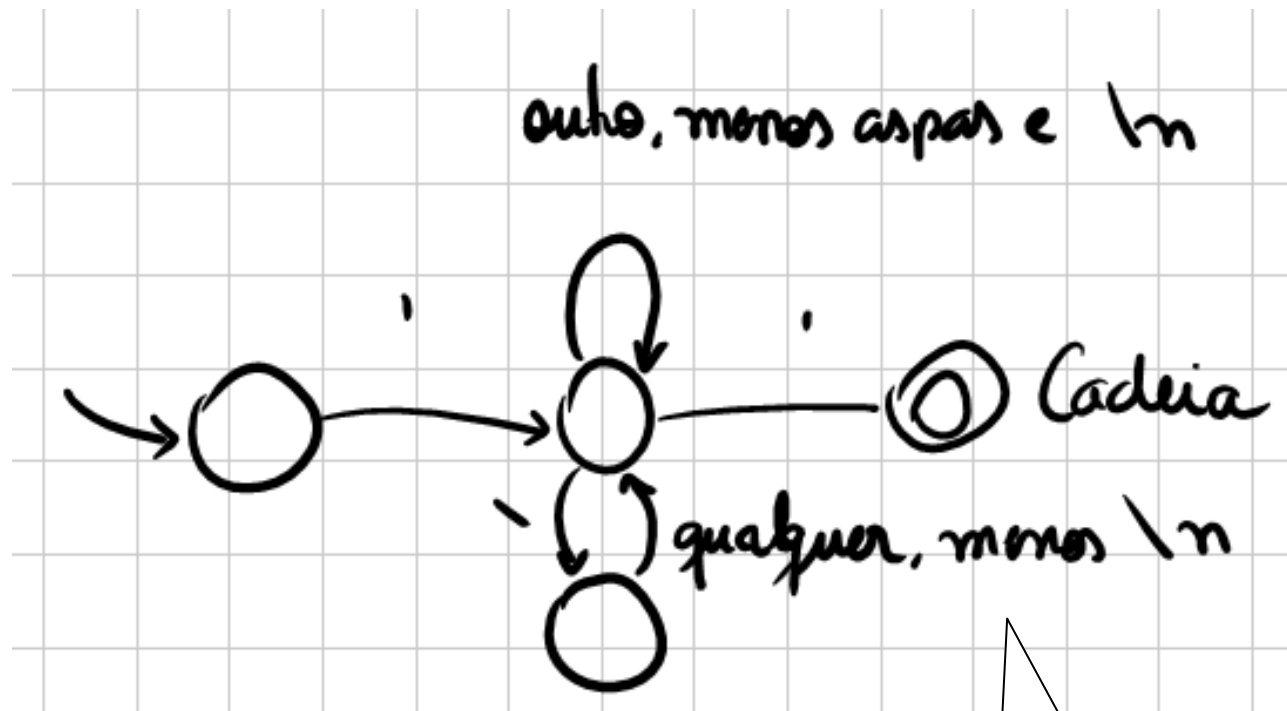
Resposta



Exercício

- 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

Resposta

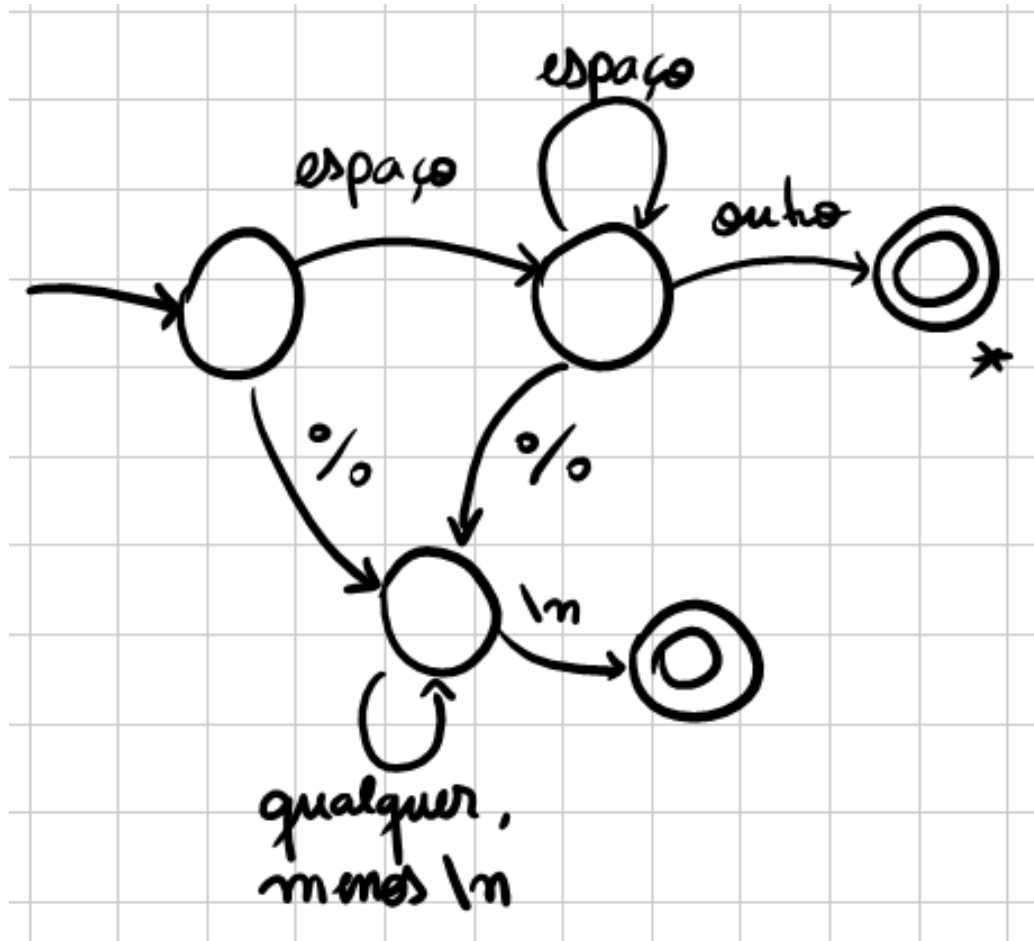


Note que aqui não é necessário retroceder

Exercício

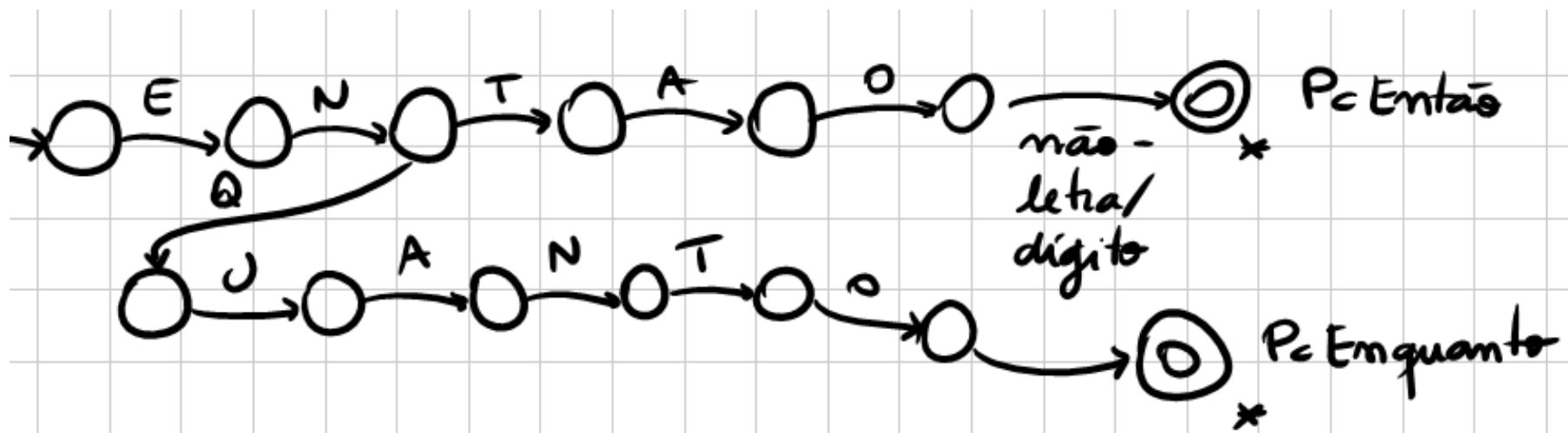
- Comentários (tudo entre % e \n)
- Espaços em branco (incluindo \n, \t, etc)
- Obs: faça em um único diagrama

Resposta



Palavras-chave

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



Implementação

- Precisaremos de um método para “zerar” o lexema atual
 - E outro para “confirmar”
- Depois implementamos cada padrão em um método diferente
- Definimos uma cascata de regras
 - Ordem lógica (Ex: identificadores depois de palavras-chave)
- O método proximoToken testa regra a regra, na ordem definida
 - Para cada padrão:
 - Caso seja bem sucedido, “confirmar”
 - Caso não seja, “zerar”
- Vamos tentar?

Demonstração

Implementação

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

Demonstração

Implementação

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

Outra opção

- Testar todas as regras “simultaneamente”
- A cada leitura de caractere, testar todas as regras
 - Fazer a correspondência com a mais longa cadeia reconhecida
- Mas ainda tem problema na manutenção do código

Opção 3

- A preferida
- Só para quem fez LFA e sabe que é possível
 - Especificar padrões usando Expressões e Definições Regulares
 - Converter ER \rightarrow NFA \rightarrow DFA automaticamente
 - Minimizar estados para aumentar eficiência

Opção 3

- Como seria:
- Especificamos:
 - Letra \rightarrow [a-zA-Z] { código }
 - Dígito \rightarrow [0-9] { código }
 - Variável \rightarrow Letra(Letra|Dígito)* { código }
- E um sistema automático geraria uma implementação
 - Criação instantânea
 - Manutenção facilitada
- Vamos ver como seria

Demonstração

Geradores de analisadores léxicos

Teoria e conceitos

Especificação de tokens

- Na prática, expressões regulares são a melhor opção
 - Não dá para representar tudo, mas para análise léxica é mais do que suficiente
 - Mais importante, possibilita implementação automática

Expressões regulares

- Descrevem linguagens regulares
 - Alfabeto: conjunto de símbolos possíveis
 - $\{0,1\}$, $\{a,b\}$, unicode, ASCII
 - Cadeia/palavra/string: sequência finita de símbolos extraídos do alfabeto
 - Linguagem: conjunto de cadeias sobre um alfabeto
 - Linguagem regular:
 - Tipo especial de linguagem
 - Associado aos autômatos finitos determinísticos/não-determinísticos
 - Linguagem formada pela aplicação de **operações regulares** sobre outras linguagens regulares

Operações regulares

- Concatenação
 - $LM = \{st \mid s \text{ está em } L \text{ e } t \text{ está em } M\}$
 - $L \cup M = \{s \mid s \text{ está em } L \text{ ou } s \text{ está em } M\}$
 - $L^* = \text{União de todas as possíveis palavras de } L$
 - $L^+ = L^* - \{\epsilon\}$
- Exercício (para lembrar)
 - Seja $L = \{A,B,C,\dots,Z,a,b,\dots,z\}$
 - Seja $D = \{0,1,2,\dots,9\}$
- Defina:
 - $L \cup D$
 - LD
 - L^*
 - $L(L \cup D)$
 - D^+

Respostas

- $L \cup D = \{A, B, C, \dots, Z, a, b, c, \dots, z, 0, 1, 2, \dots, 9\}$
- $LD = 520$ cadeias de tamanho 2
 - $\{A0, A1, A2, \dots, A9, B0, B1, \dots, B9, \dots, z0, z1, \dots, z9\}$
- $L^* =$ Conjunto de todas as cadeias de letras, incluindo a vazia
- $L(LUD)^* =$ 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
- $(r)|(s) = L(r) \cup L(s)$ (união)
- $(r)(s) = L(r)L(s)$ (concatenação)
- $(r)^* = (L(r))^*$ (fecho/estrela Kleene)
- $(r)^+ = (L(r))^+$ (fecho positivo)
- $(r) = L(r)$ (parêntesis servem para agrupar)

Expressões regulares

- Exercício - descreva as seguintes linguagens:
 - $a|b \rightarrow \{a,b\}$
 - $(a|b)(a|b) \rightarrow \{aa,ab,ba,bb\}$
 - $a|a^*b \rightarrow \{a,b,ab,aab,aaab,aaaab,\dots\}$
 - $a(a|b)^+ \rightarrow$ todas as cadeias que começam com a

Definições regulares

- Regras auxiliares
 - Conveniência notacional
 - Consiste em dar nomes a algumas expressões comuns e usá-los em outras expressões
 - Não pode haver recursividade!
- Ex:
 - Letra $\rightarrow a \mid b \mid c \dots \mid z$
 - Dígito $\rightarrow 0 \mid 1 \mid 2 \mid \dots \mid 9$
 - Id $\rightarrow \text{Letra} (\text{Letra} \mid \text{Dígito})^*$

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)
- Ex:

```
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
- Ex:

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

```
Digito : '0'..'9';
```

Expressões regulares no ANTLR

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

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

Id: Letra (Letra|Digito|'_') *;

Cadeia : '"' (~ ('\\' | '"')) * '"';

Tudo : .+;

Expressões regulares no ANTLR

- Palavra-chave *fragment* denota definições regulares auxiliares
- Ex:

```
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 houverem duas ou mais regras que correspondem a um mesmo número de caracteres de entrada, aquela que aparece primeiro no programa é escolhida

Ambiguidade

- Ex:

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 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)
 - Exemplo: `'\''.*'\''` (sequência de caracteres envoltos em aspas)
- Dada a entrada

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



`'\''.*'\''`

Sequência reconhecida: `'primeira'`

Ambiguidade

- Ou seja: ANTLR é ganancioso com todas as regras, exceto `.*` e `.+`
- Mas ainda é necessário cuidado
 - Exemplo: `'\'' (~ ('\n')) * '\''` (sequência de caracteres, com exceção de fim de linha, envoltos em aspas)
- Neste caso o ANTLR será ganancioso, e irá gerar um analisador que “consume” as aspas sem se preocupar
 - Ou seja, o analisador não vai funcionar como deveria

Ambiguidade

- Alternativa 1: definir que aquela regra não deve utilizar comportamento ganancioso
 - `'\'' (options {greedy=false;} : ~('\n')) * '\''`
- Alternativa 2: definir explicitamente a condição de parada
 - Ou transformar a regra para remover o não-determinismo
 - `'\'' (~('\'' | '\n')) * '\''`

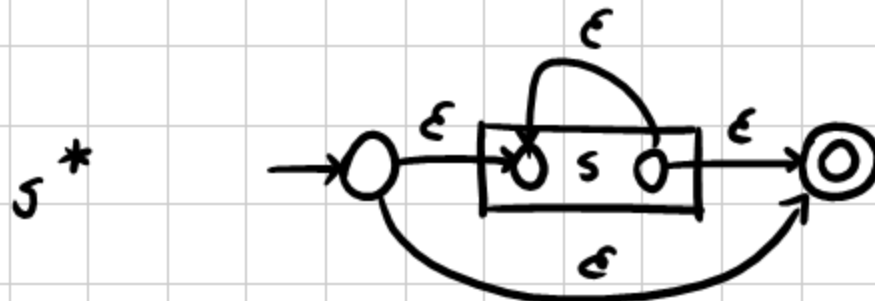
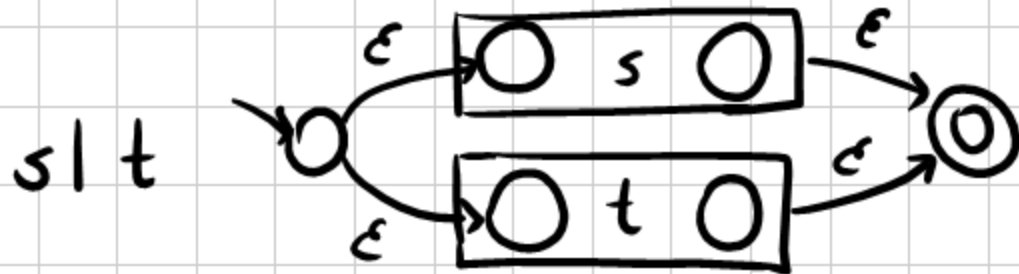
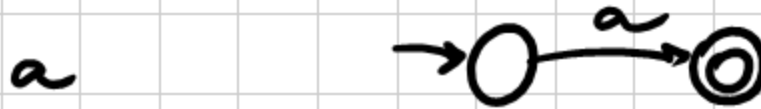
Exercícios

- 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 \rightarrow NFA
 - Algoritmo de McNaughton-Yamada-Thompson



A ciência por trás

- Notação estendida para ERs:
 - $E^+ = EE^*$
 - $E? = (E \mid \varepsilon)$
 - $'a'..'z' = 'a' \mid 'b' \mid 'c' \mid \dots \mid 'z'$
 - $\sim('a'..'c') = 'd' \mid 'e' \mid \dots$ (todos símbolos do alfabeto)

A ciência por trás

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

A ciência por trás

- Ao invés de converter NFA para DFA, é possível executar o NFA 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 DFA
 - Mais eficiente do que usando um NFA 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

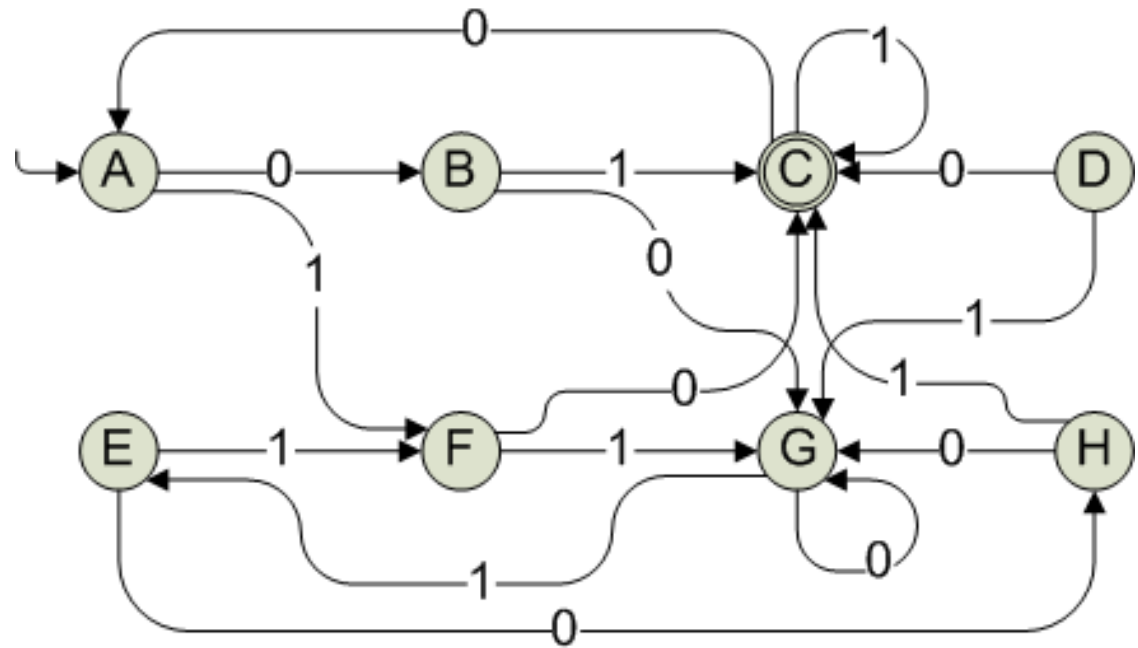
A ciência por trás

- Minimização de estados
 - Sempre existe um DFA 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)

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
	A	B	C	D	E	F	G



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') +;

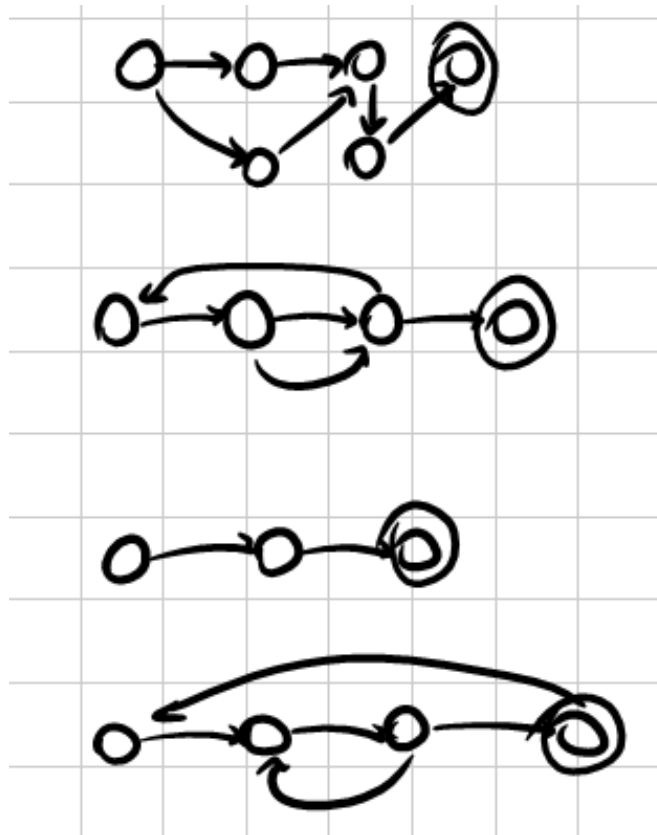
NUMREAL : ('+' | '-')? ('0' .. '9') + ('.' ('0' .. '9') +)?;

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

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

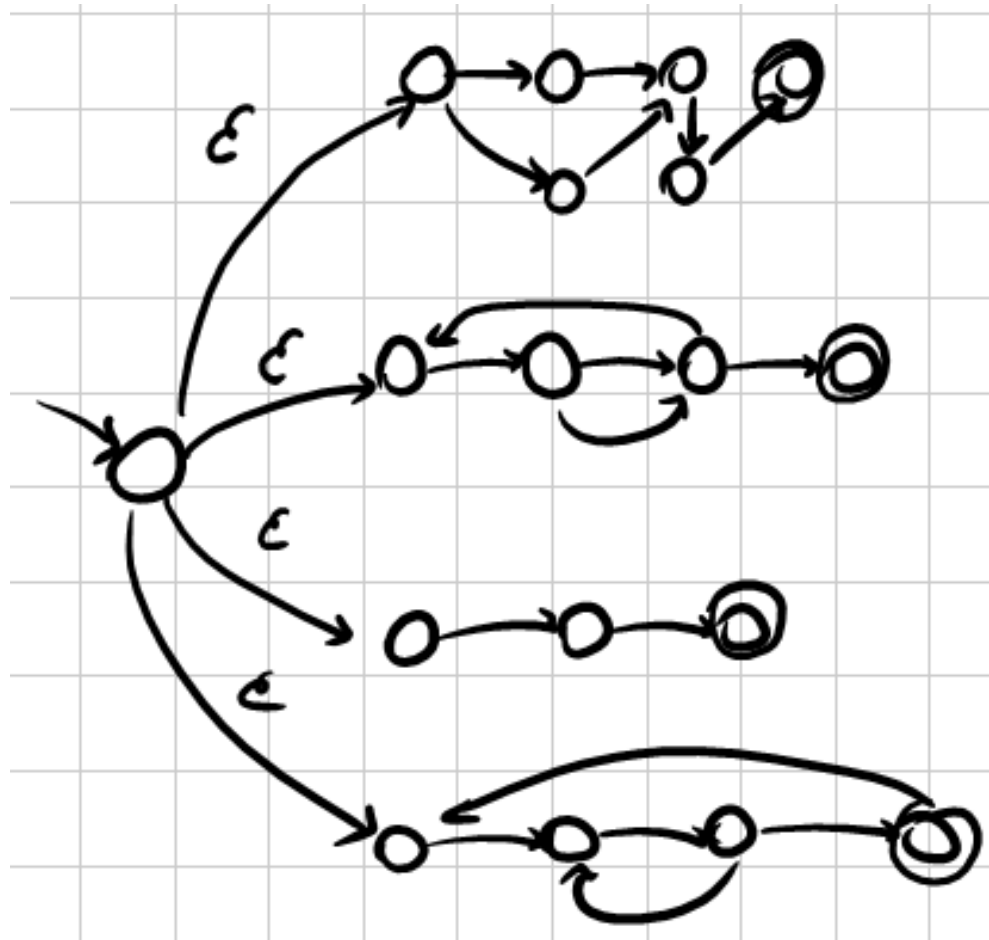
Estratégia para geradores de analisadores léxicos

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



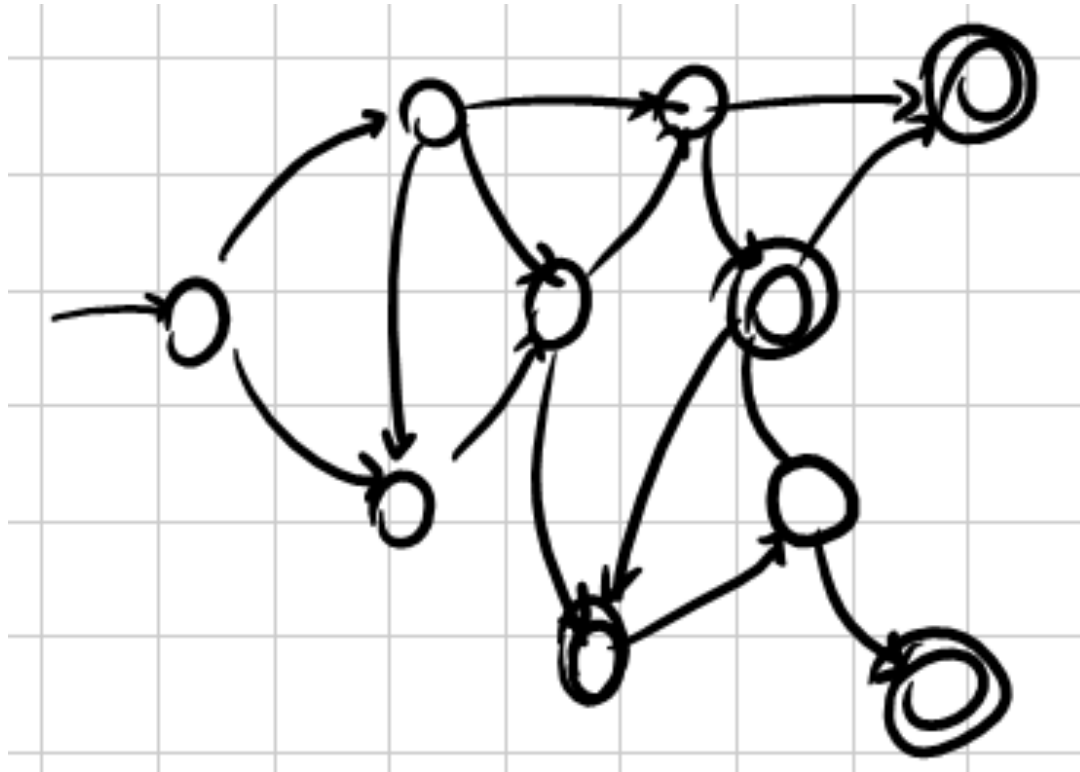
Estratégia para geradores de analisadores léxicos

- 3. Combinar os NFAs individuais em um único



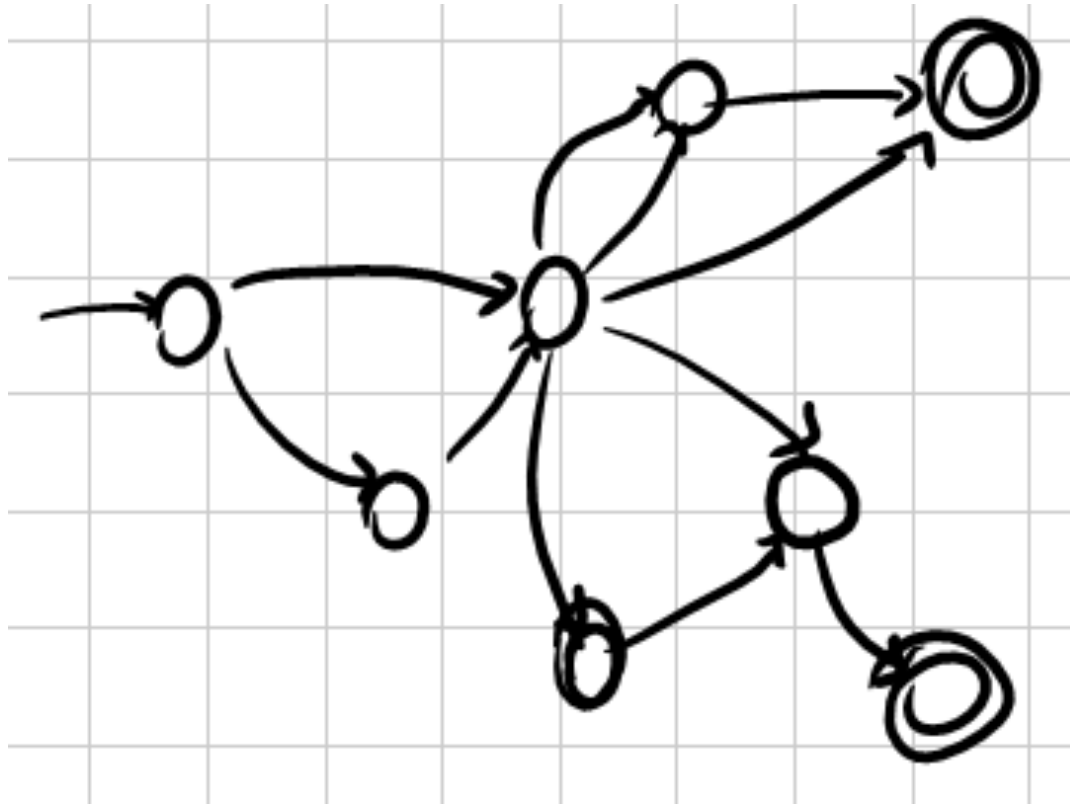
Estratégia para geradores de analisadores léxicos

- 4. Converter o NFA em um DFA



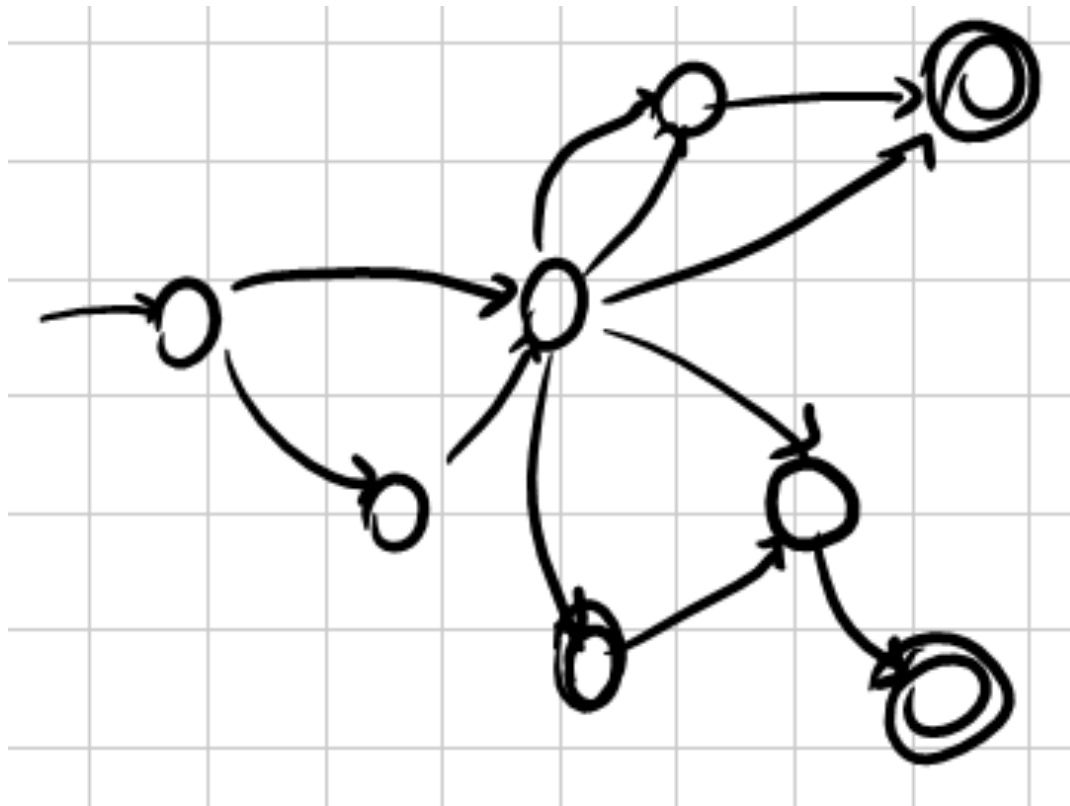
Estratégia para geradores de analisadores léxicos

- 5. Minimizar os estados do DFA



Estratégia para geradores de analisadores léxicos

- 6. Executar o DFA
 - 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 idéia 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