Construção de Compiladores

Prof. Dr. Daniel Lucrédio
DC - Departamento de Computação
UFSCar - Universidade Federal de São Carlos

Tópico 02 - Análise Léxica

Referências bibliográficas

Alfred V. Aho, Monica S. Lam, Ravi Sethi, Jeffrey D. Ullman. Compiladores: Princípios, Técnicas e Ferramentas (2a. edição). Pearson, 2008.

Kenneth C. Louden. Compiladores: Princípios E Práticas (1a. edição). Cengage Learning, 2004.

Terence Parr. The Definitive Antlr 4 Reference (2a. edição). Pragmatic Bookshelf, 2013.

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

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

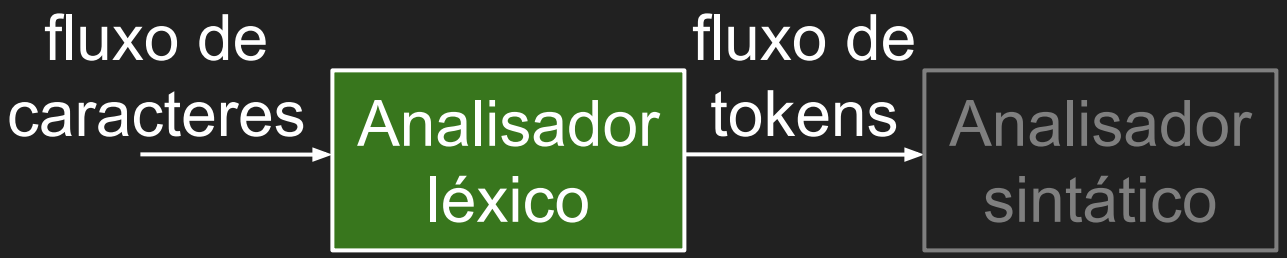
- Nomes das coisas
- É o que mais facilmente emerge na consciência dos locutores
- Quem tem bebês sabe como eles aprendem a falar

- Ações
- Composição
- Conceitos complexos

Contexto

- 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

O analisador léxico



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:

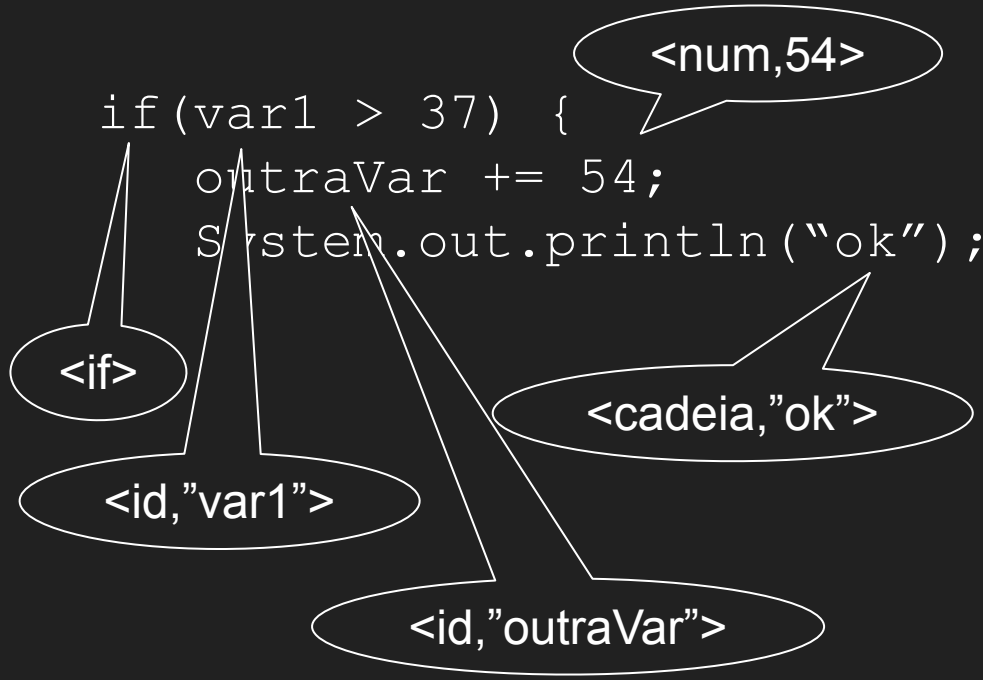
```
class Token {
    TipoToken tipo;
    String valor;
}
```

<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
DECLARACOES, 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

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

```
<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”> <EOF>
```

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

Erros léxicos

- Erros simples podem ser detectados
  - Ex
- Mas pa inho
  - Po
    wh
  - É sintaxe?
- A culpa
  - O tudo se
    en
- Em ou palavras, quase sempre tem um padrão que reconhece o lexema

Erros léxicos somente aparecem quando não existe um padrão capaz de reconhecer a entrada!

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
LER argumento
ATRIBUIR argumento A fatorial
SE argumento == 0 ENTAO
 ATRIBUIR 1 A Fatorial
ENQUANTO argumento > 1
 INICIO
  ATRIBUIR fatorial * (argumento
           A fatorial
  ATRIBUIR argumento - 1 A argume
 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

Como implementar?

Para “sentir o drama”

- Vamos tentar implementar um analisador léxico para a linguagem ALGUMA

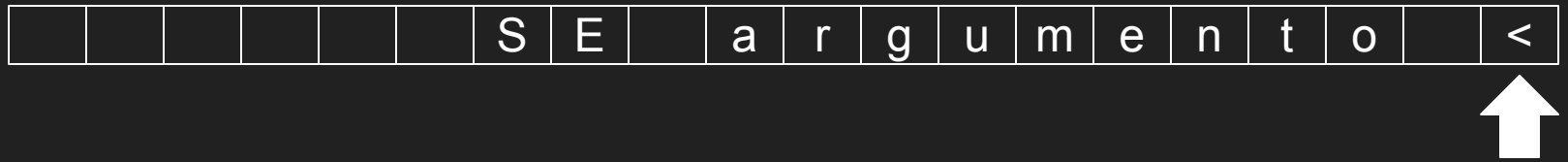
Demonstração 1

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

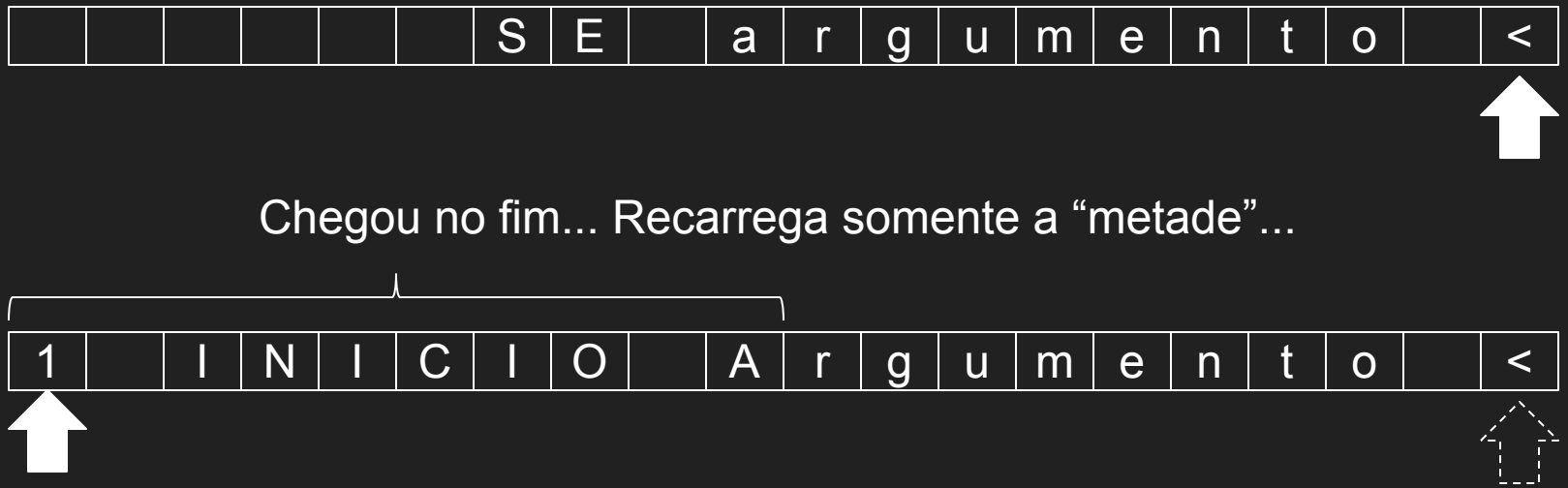


Chegou no fim... recarregando...

Para retroceder, é necessário recarregar o buffer anterior!

Buffer duplo

- Portanto, é comum o uso de um buffer duplo



Demonstração 2

Continuando

- Como traduzir os padrões em código?

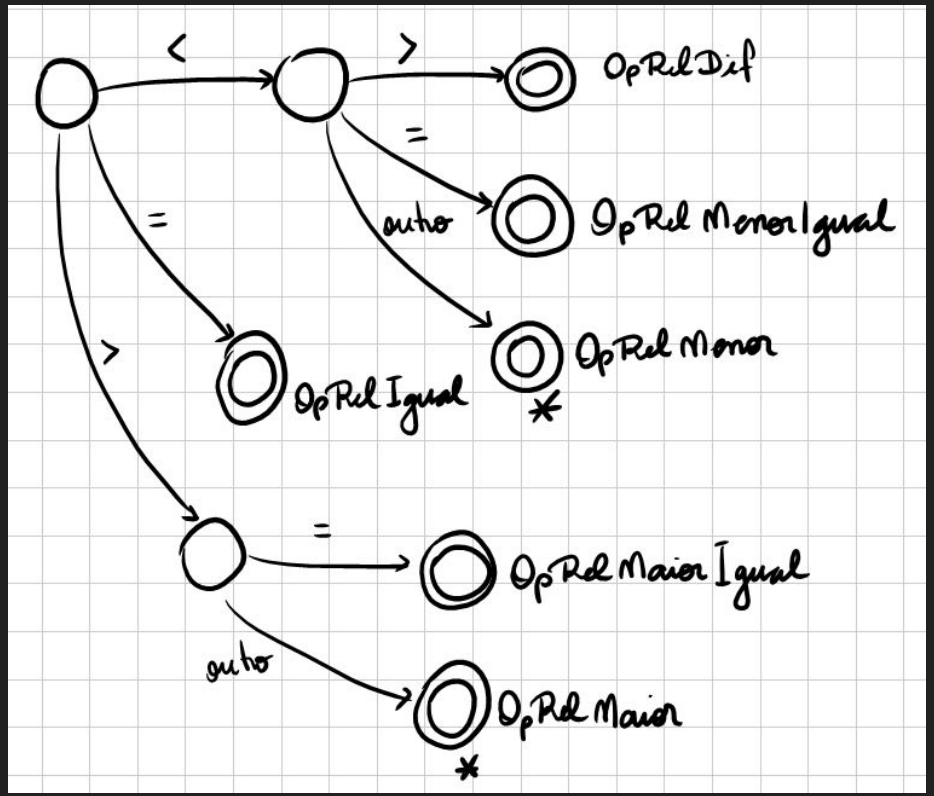
Padrão	**Sigla**
DECLARAÇÕES, ALGORITMO, INTEIRO, REAL, ATRIBUIR, LER, IMPRIMIR, SE, ENTAO, ENQUANTO, INICIO, FIM	3 primeiras letras
*, /, +, -	OpArit
<, <=, >=, >, =, <>	OpRel
E, OU	OpBool
:	Delim
(,)	AP / FP
Seq. de letras ou números que começam com letra minúscula	Var
Sequências de dígitos (sem vírgula)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

Reconhecimento de padrões

- Opção 1
  - Basta implementar a lógica utilizando nossa criatividade
  - E ferramentas disponíveis na maioria das LP
  - Diagramas podem ajudar

Diagramas de transição

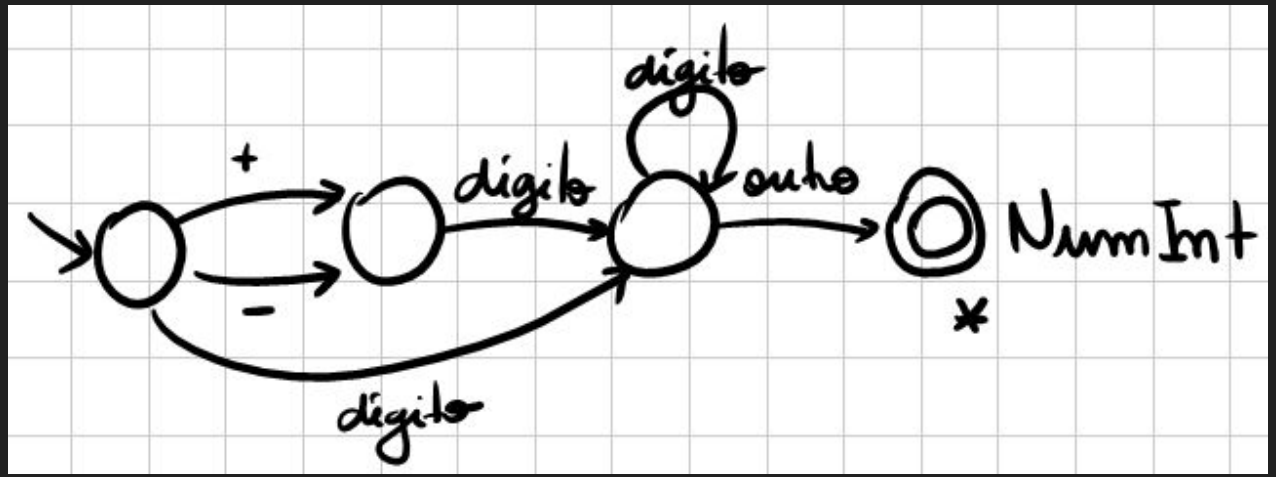
- Diagrama de estados
  - Modelo visual que facilita a implementação da lógica do reconhecimento
- Autômato finito determinístico
  - Com indicação de retrocesso no final


<
>
OpRelDif
=
outro
OpRelMenorIgual
=
OpRelMenor
>
OpRelIgual
*
=
OpRelMaiorIgual
outro
OpRelMaior
*

Exercício

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

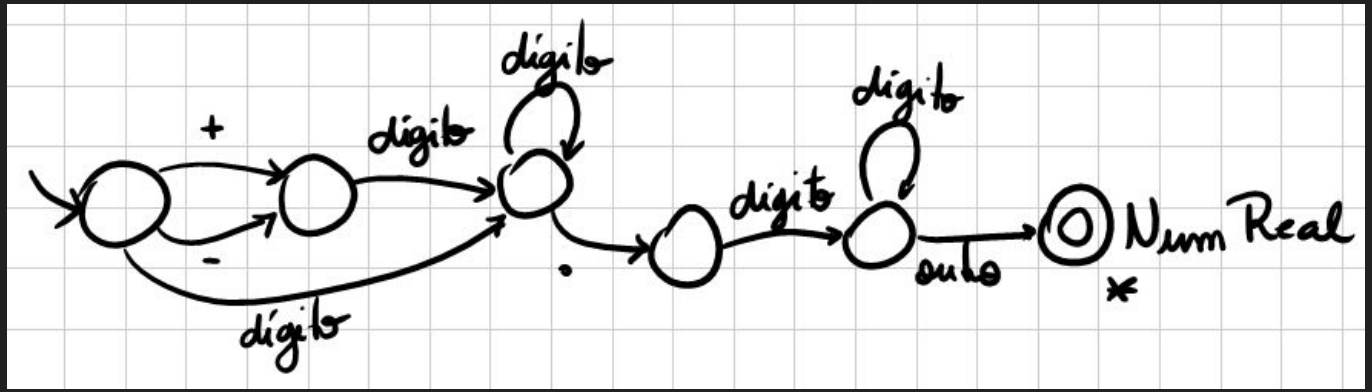
Resposta



Exercício

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

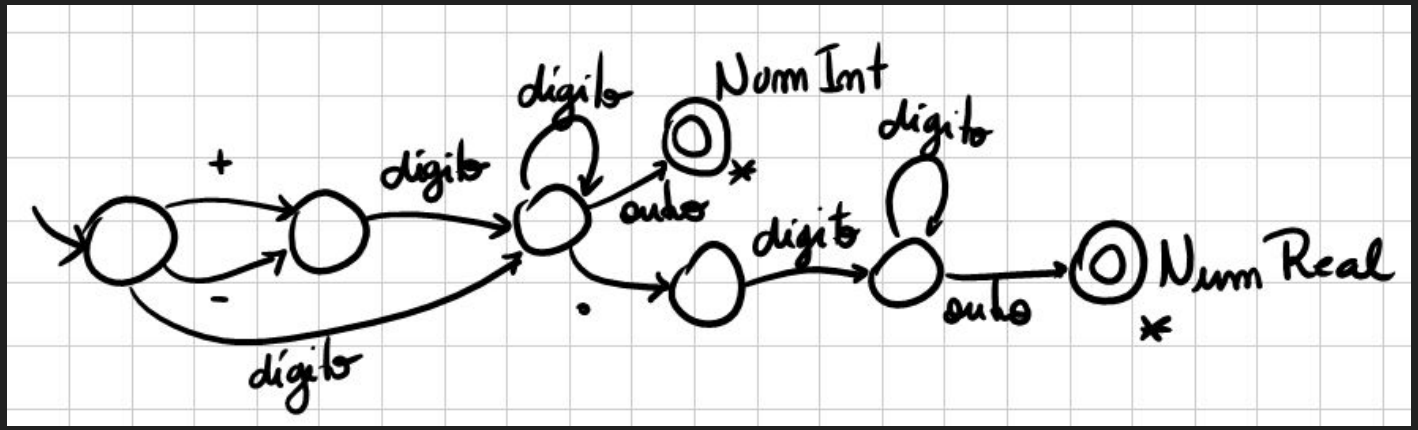
Resposta



Exercício

- Junte ambos os diagramas em um só

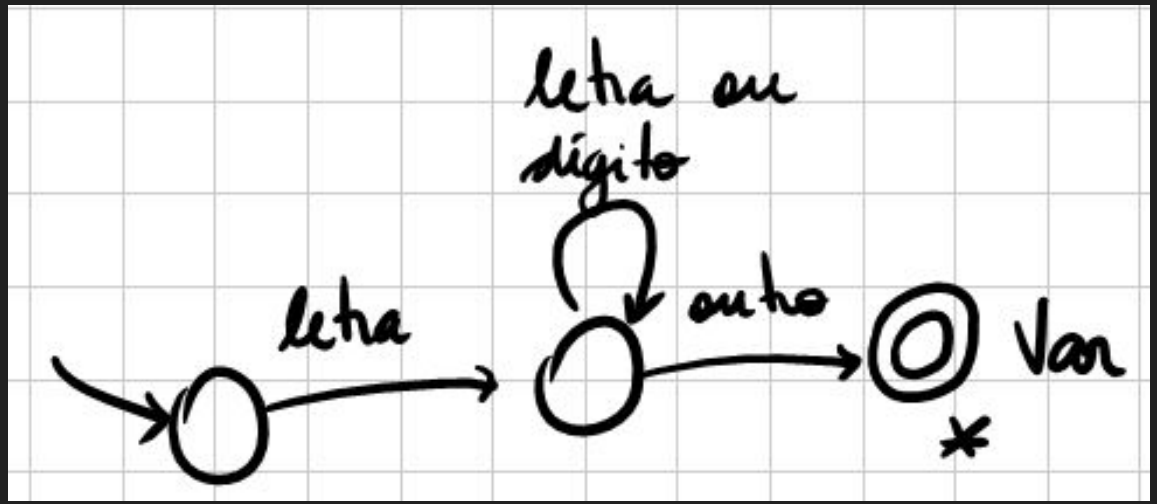
Resposta



Exercício

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

Resposta



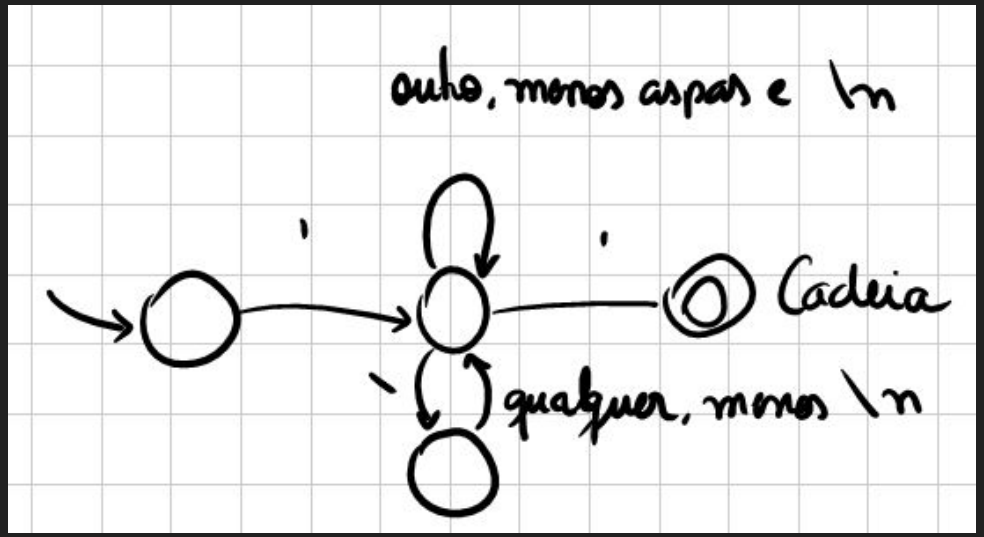
Exercício

- 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

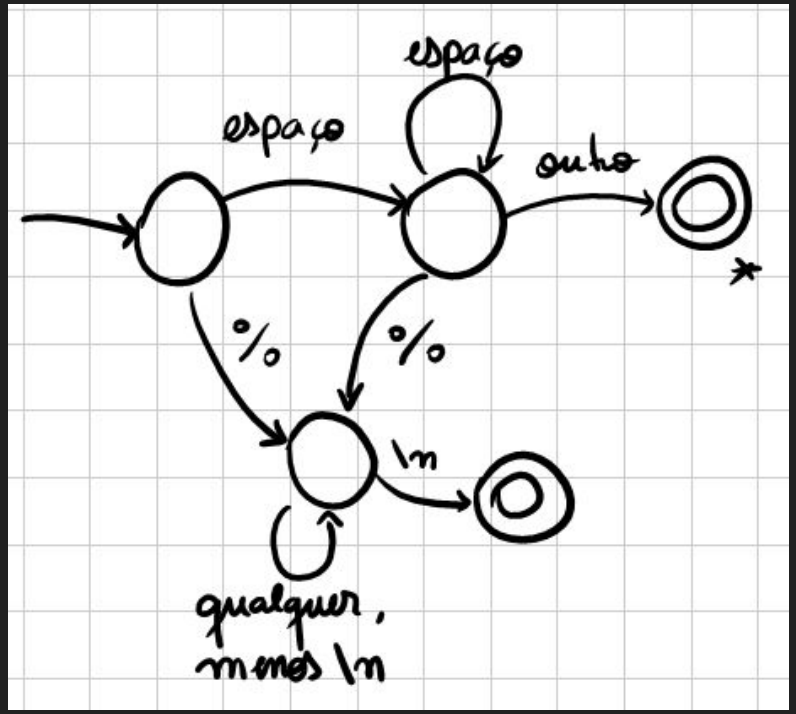


Note que aqui não é necessário retroceder

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



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 3

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

Opção 2 (preferida)

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

Opção 2 (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

Demonstração 4

Expressões regulares no ANTLR

- Operações regulares
  - união, concatenação, fecho…
- Definições regulares
  - regras auxiliares, sem recursividade
- 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

E se eu criar as regras assim?

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

Se entrada == “integers”

Ambas aceitam a entrada

A regra 1 é escolhida, pois engloba os 8 caracteres

Ou seja, nunca será escolhida a regra 2!

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

E se eu criar as regras assim?

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

Lembre-se deste exemplo!

Se entrada == “integers”

Ambas aceitam a entrada

A regra 1 é escolhida, pois engloba os 8 caracteres

Ou seja, nunca será escolhida a regra 2!

Se entrada == “integer”

Ambas aceitam a entrada

A regra 1 é escolhida, pois ambas englobam os 7 caracteres, mas a regra 1 aparece primeiro

Demonstração 5

Alguns geradores de analisadores léxicos

- Na disciplina usaremos o ANTLR
  - Facilidade de utilização
  - 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

Fim