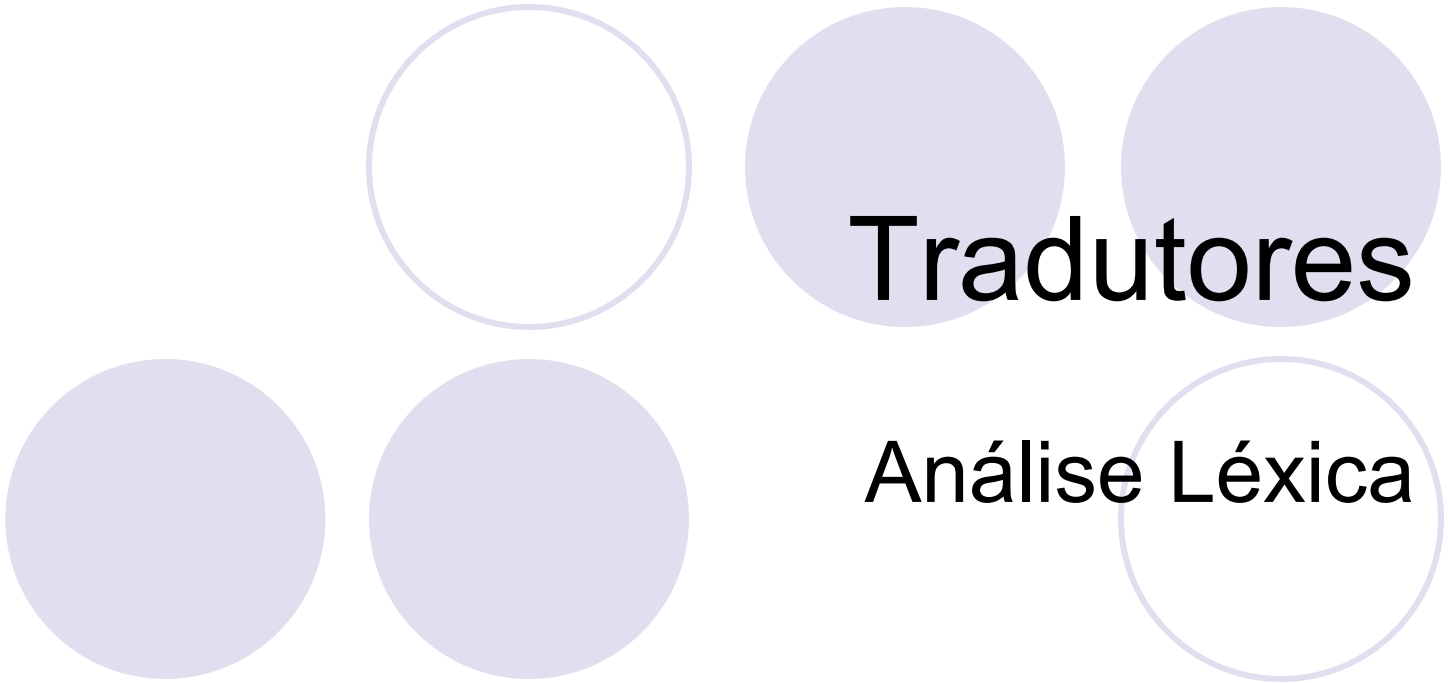


Tradutores

Análise Léxica

Sumário

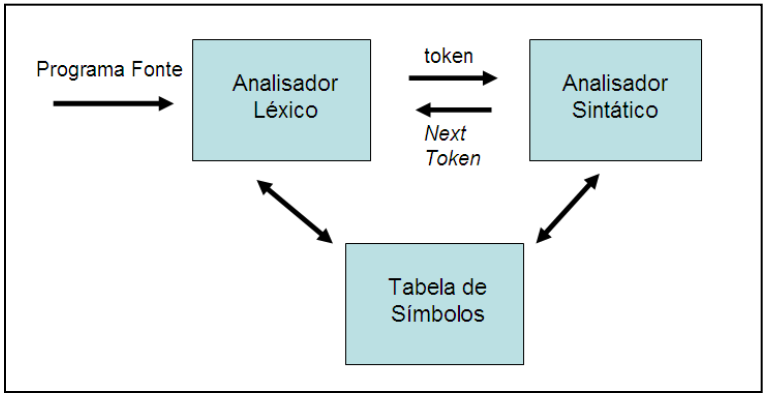
- Análise Léxica
  - Introdução
  - Análise Léxica x Análise Sintática
  - Tokens x padrões x lexemas
  - Erros Léxicos
  - *Bufferização* da entrada
  - Especificação de *Tokens*

Sumário

- Expressões regulares
- Definições regulares
- Especificação de tokens
- Reconhecimento de tokens

Análise Léxica

- Introdução
  - O papel do analisador léxico é ler os caracteres da entrada (programa fonte), agrupa-los em lexemas e produzir como saída uma seqüência de *tokens*
  - Pode ser dividido em dois processos:
    - Scanning
      - Retira espaços em branco e comentários
    - Análise Léxica
      - Produz a seqüência de *tokens*



Análise Léxica X Análise Sintática

- Diferentes razões para separar a análise léxica da análise sintática
  - Simplicidade de projeto
    - Simplifica o desenvolvimento do analisador sintático. Ex.: Este não precisa lidar com espaços em branco
  - Eficiência
    - Permite que técnicas específicas de otimização sejam aplicadas ao analisador léxico
  - Portabilidade
    - Peculiaridades específicas são tratadas no analisador léxico. Ex.: caracteres de nova linha

Tokens, padrões e lexemas

- *Token*
  - Par ordenado formado pelo nome do *token* e atributos (opcionais ou não). Nome do *token* é um símbolo abstrato
- Padrão
  - descrição da forma que um token pode assumir. Normalmente é dado por uma expressão regular.
- Lexema
  - sequencia de caracteres do programa fonte reconhecida pelo padrão. É identificado pela analisador léxico como uma instância do token

Tokens, padrões e lexemas

- Exemplo

```
printf("Total = %d\n", total);
```

Token	Lexema
reserved_word	printf
l_paren	(
string_literal	"Total = %d\n"
comma	,
id	total
r_paren	)
semicolon	;

Exercício

- Divida o seguinte código escrito em C em tokens:

```
float limitedSquare(x) float x {
          /* returns x-squared, but never more than 100 */
       return(x <= -10.0 || x >= 10.0)?100:x*x;
}
```

Desafios da Análise Léxica

- FORTRAN
  - Espaços em branco são considerados parte da especificação léxica
  - VAR1 é o mesmo que VA R1

```
DO 5 I = 1,25
```

```
DO 5 I = 1.25
```

Desafios da Análise Léxica

- Syntaxe de templates em C++
  - Foo<Bar>
- Syntaxe de streams em C++
  - cin >> var;
- Templates aninhados
  - Foo< Bar<Bazz**>>**

Tokens, padrões e lexemas

- Classes de *tokens* (comum a diferentes linguagens de programação)
  - Um token para cada palavra reservada
  - Um token para cada operador
  - Um token representando todos os identificadores
  - Um ou mais tokens representando constantes tais como constantes numéricas ou cadeias de caracteres
  - Um token para cada símbolo de pontuação, como (e), ;, :.

Atributos de *Tokens*

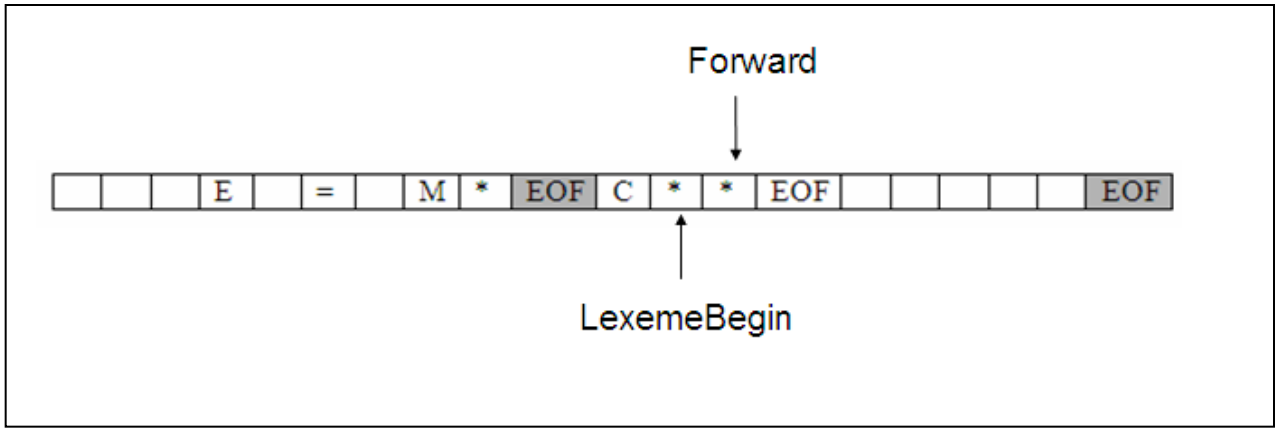
- E = M * C ** 2
  - **<id, ponteiro para a entrada de E na tabela de símbolos>**
  - **<operador_atribuição>**
  - **<id, ponteiro para a entrada de M na tabela de símbolos>**
  - **<operador_multiplicação>**
  - **<id, ponteiro para a entrada de C na tabela de símbolos>**
  - **<operador_exponenciação>**
  - **<número, valor inteiro 2>**

Erros Léxicos

- Dentre as fases envolvidas no processo de tradução, a análise léxica é a que detecta o menor número de erros
  - Ex: `fi  (a == f(x))`
  - Este erro será detectado em uma fase subseqüente do processo (análise sintática)

Bufferização da Entrada

- *LexemeBegin*
  - Marca o início do *token sendo reconhecido*
- *Forward*
  - *Marca o caractere corrente sendo analisado.*
  - *É incrementado a cada novo caractere analisado.*
  - *Para de ser incrementado quando o lexema é reconhecido*



Especificação de *Tokens*

- Alfabeto
  - Conjunto finito de símbolos
    - Letras, dígitos
    - {0,1} → alfabeto binário
    - ASCII
    - Unicode
- Sentença sobre um alfabeto
  - Seqüência finita de símbolos obtida a partir do alfabeto

Especificação de *Tokens* (cont.)

- O tamanho de uma seqüência *s* é denotado por |*s*|
- A seqüência vazia é denotada por ε e tem tamanho zero ((|ε | = 0)
- Linguagem
  - Qualquer conjunto contável de seqüências pertencentes a um alfabeto finito
  - Definição ampla
    - Linguagem vazia {0}
    - Linguagem {ε}
    - Conjunto de todos os programas sintaticamente corretos escritos em C

Especificação de *Tokens* (cont.)

- Se *x* e *y* são sentenças, então a concatenação de *x* e *y* é dada por *xy*
  - Ex.: *x* = *ban, y* = *ana*
  - *xy* = *banana*
- Exponenciação
  - Seja a sentença *s*.
    - $s^0 = \varepsilon$, $s^2 = ss$, $s^3 = sss$

Especificação de *Tokens* (cont.)

- Operações sobre linguagens

Operação	Definição
União de L e M	$L \cup M = \{s \mid s \text{ está em } L \text{ ou } s \text{ está em } M\}$
Contatenação de L e M	$LM = \{st \mid s \text{ está em } L \text{ e } t \text{ está em } M\}$
Fechamento de L (Kleene)	$L^* = \cup_{i=0}^{\infty} L^i$
Fechamento Positivo de L	$L^+ = \cup_{i=1}^{\infty} L^i$

Especificação de *Tokens* (cont.)

- Seja *L* o conjunto de letras {A,B,…Z,a,b,…z}
- Seja *D* o conjunto de dígitos {0.1….9}
- Tanto *L* quanto *D* podem ser vistos de duas maneiras diferentes
  - Alfabetos de letras maiúsculas e minúsculas
  - Linguagens, onde todas as sequencias possuem tamanho 1

Especificação de *Tokens* (cont.)

- Tomando-se *L* e *D* como linguagens, a partir dos operadores de união, concatenação, fechamento e fechamento positivo, as seguintes linguagens podem ser obtidas através de *L* e *D*

1. $L \cup D$ é o conjunto de todas letras e dígitos.
2. LD é o conjunto de 520 sentenças, cada uma delas com 2 caracteres.
3. L^4 é o conjunto de todas as sentenças compostas de 4 letras
4. L^* é o conjunto de todas as letras, incluindo a sentença vazia
5. $LU(LUD)^*$ é o conjunto de todas as sentenças formadas por letras e dígitos e que iniciam com uma letra
6. D^+ é o conjunto de todas as sentenças formadas por um ou mais dígitos.

Expressões Regulares

- Linguagem denotada pelo ítem 5 pode ser utilizada para representar os identificadores de uma linguagem de programação.
  - União
  - Concatenação
  - Fechamento
- Processo utilizado como base para a criação de expressões regulares
  - Notação utilizada para a representação de todas as linguagens que podem ser construídas através da aplicação destes operadores em símbolos de algum alfabeto

Expressões regulares

- “Método formal para especificar um padrão de texto”
- As regras que definem as expressões regulares sobre um alfabeto ∑ são definidas da seguinte maneira:

BASE: Duas regras formam a base:

1. ε é uma expressão regular, e *L(*ε) = {ε}, isto é, a linguagem na qual o único membro é a sentença vazia.

2. Se *a* é um símbolo em ∑ , então *a* é uma expressão regular, e *L(a)* = {*a*}, isto é, a linguagem com apenas uma sentença de tamanho 1 onde o único símbolo é *a*.

Expressões regulares

INDUÇÃO: Dadas duas expressões regulares *r* e *s* denotando respectivamente as linguagens *L(r)* e *L(s)*.

1. (*r*) |(*s*) é uma expressão regular denotando a linguagem *L*(*r*) U *L*(*s*).
2. (*r*)(*s*) é uma expressão regular denotando a linguagem *L*(*r*)*L*(*s*)
3. (*r*)* é uma expressão regular denotando a linguagem (*L*(*r*))*
4. (*r*) é uma expressão regular denotando *L*(*r*). Parênteses podem ser adicionados em expressões regulares sem alterar a linguagem denotada.

Precedência:

1. O operador unário * possui a maior precedência e é associativo à esquerda
2. A concatenação possui a segunda maior precedência e é associativa à esquerda
3. | possui a menor precedência e é associativo à esquerda

Expressões Regulares

- Exemplo

  Seja o alfabeto ∑ = {*a,b*}

  1. A expressão regular *a*|*b* denota a linguagem {*a,b*}

  2. (*a*|*b*)(*a*|*b*) denota { *aa, ab, ba, bb* }, a linguagem de todas as sentenças de tamanho 2 sobre o alfabeto ∑.

  3. *a** denota a linguagem que consiste de todas as sentenças com zero ou mais *a*. Por exemplo {*a,aa,aaa,.....*}

  4. (*a*|*b*)* denota o conjunto de todas as sentenças que consistem de zero ou mais instâncias de *a* ou *b*

  5. *a*|*a***b* denota a linguagem {*a,b,ab,aab,aaab,...*}, isto é, a sentença *a* e todas as sentenças que consistem de zero ou mais *a* s terminando com um *b*.

Expressões Regulares

Uma linguagem que pode ser definida por uma expressão regular é chamada de conjunto regular. Se duas expressões regulares *r* e *s* denotam o mesmo conjunto regular, diz-se que elas são equivalentes (*r* = *s*).

Diversas leis algébricas se aplicam a expressões regulares:

Lei	Descrição
$r\|s = s\|r$	| é comutativa
$r\|(s\|t) = (r\|s)\|t$	| é associativa
$r(st) = (rs)t$	A concatenação é associativa
$r(s\|t) = rs\|rt\ ;\ (s\|t)r = sr\|st$	A concatenação é distributiva sobre a |
$\varepsilon r = r\varepsilon = r$	ε é a identidade para a concatenação
$R^* = (r\|\ \varepsilon)^*$	ε é garantido na operação de fechamento
$r^{**} = r^*$	* é idempotente

Definições regulares

- Notação utilizada para representar expressões regulares através de definições
- Uma vez definidas podem ser combinadas de modo a formar expressões regulares mais complexas
- Se ∑ é um alfabeto de símbolos básicos, então uma definição regular é uma seqüência de definições da forma:

$$d_1 \rightarrow r_1$$
$$d_2 \rightarrow r_2$$
$$d_3 \rightarrow r_3$$
$$\vdots$$
$$d_n \rightarrow r_n$$

Definições regulares

• Cada *di* é um novo símbolo não contido em ∑ e diferente de outras definições
• Cara *ri* é uma expressão regular sobre o alfabeto ∑ U {*d1, d2, ..., dn-1*}

$d_1 \rightarrow r_1$
$d_2 \rightarrow r_2$
$d_3 \rightarrow r_3$
.
$d_n \rightarrow r_n$

letra	→	A|B|C...| Z| a|b|...|z| _
dígito	→	0|1|2|...|9
id	→	*letra*(*letra*|*dígito*)*

dígito	→	0 | 1 | ... |9
dígitos	→	*dígito dígito**
fracaoOpcional	→	. *dígitos* | ε
expoenteOpcional	→	(E (+ | - | ε) *dígitos*) | ε
número	→	*dígitos fraçãoOpcional expoenteOpcional*

Expressões Regulares

- Extensões

letra	→	[A-Za-z]
dígito	→	[0-9]
id	→	*letra*(*letra*|*dígito*)*

dígito	→	[0-9]
dígitos	→	*dígito*+
número	→	*dígitos* (. *dígitos*)? (E [+-]? *Dígitos*)?

Reconhecimento do Tokens

- A fim de reconhecer tokens em uma entrada é necessário que se expresse padrões para estes tokens (geralmente através de expressões regulares) e então se construa um programa que analise a entrada de modo a encontrar lexemas reconhecidos por estes padrões.
- Ex: fragmento de gramática para expressões condicionais.

stmt	→	if *expr* then *stmt*
	|	if *expr* then *stmt* else *stmt*
	|	ϵ
expr	→	*term* relop *term*
	|	*term*
term	→	id
	|	number

digit	→	`[0-9]`
digits	→	$digit^+$
number	→	*digits* (. *digits*)? (`E` `[+-]`? *digits*)?
letter	→	`[A-Za-z]`
id	→	*letter* (*letter* | *digit*)*
if	→	`if`
then	→	`then`
else	→	`else`
relop	→	`<` | `>` | `<=` | `>=` | `=` | `<>`

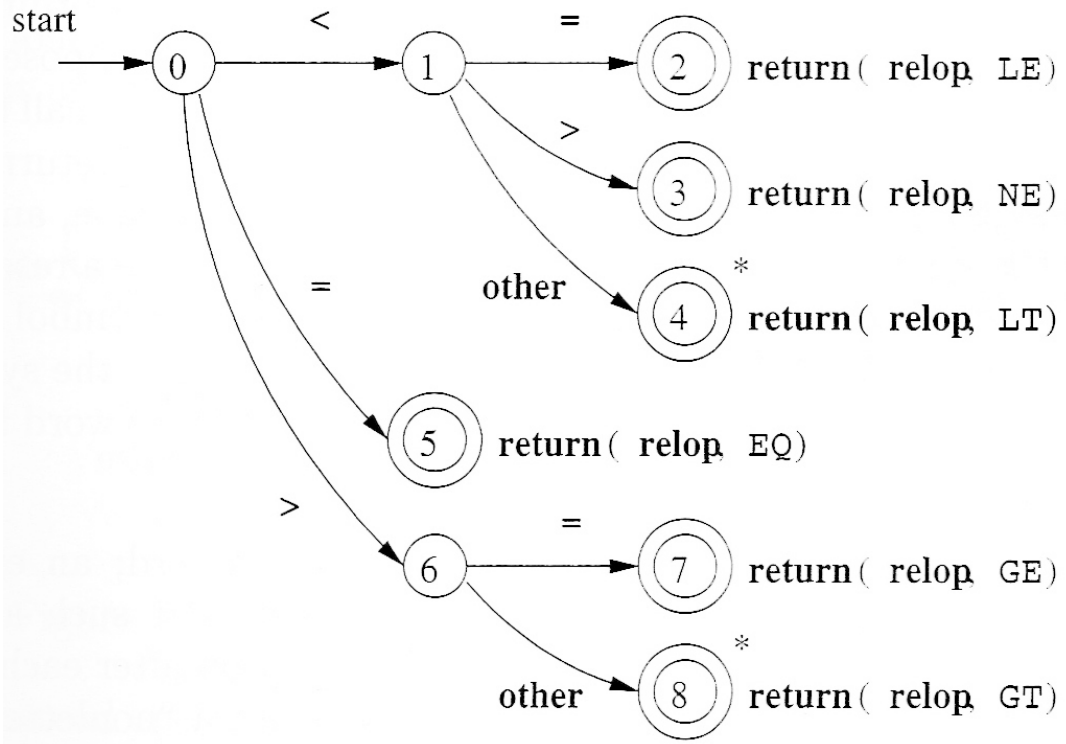
Reconhecimento do Tokens

- Lexemas, tokens e atributos correspondentes

LEXEMES	TOKEN NAME	ATTRIBUTE VALUE
Any *ws*	–	–
`if`	**if**	–
`then`	**then**	–
`else`	**else**	–
Any *id*	**id**	Pointer to table entry
Any *number*	**number**	Pointer to table entry
`<`	**relop**	`LT`
`<=`	**relop**	`LE`
`=`	**relop**	`EQ`
`<>`	**relop**	`NE`
`>`	**relop**	`GT`
`>=`	**relop**	`GE`

Reconhecimento do Tokens

- Passo intermediário
  - Diagramas de transição
    - Operadores relacionais



Reconhecimento do Tokens

- Reconhecimento de palavras reservadas
- Exercício
  - Definir diagramas de transição para identificadores e números.