Análise Léxica

Léxico

lé·xi·co (grego *leksikós*, -ê, -ón, relativo a palavras)
substantivo masculino

1. Dicionário, particularmente de língua clássica como latim ou grego.

2. [Linguística] Conjunto virtual das unidades lexicais de uma língua.

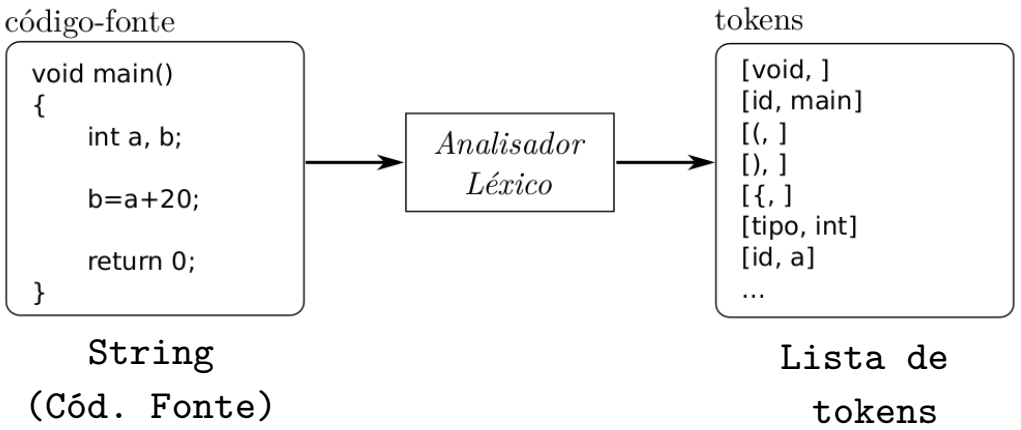
3. Compilação de palavras de uma língua = VOCABULÁRIO.

Papel do Analisador Léxico

- A palavra léxico no sentido tradicional significa algo como "*conjunto de palavras*"
- Em termos de linguagens de programação, palavras são objetos como nomes de variáveis, números, palavras-chave
- Tais palavras são tradicionalmente chamadas de *tokens*

Papel do Analisador Léxico

- Um analisador léxico, lexer ou scanner, recebe uma string de caracteres individuais e divide esta string em uma sequência de tokens
- Essa sequência de tokens é utilizado pelo analisador sintático



String (Cód. Fonte)

Lista de tokens

IMPORTANTE:
A ordem das palavras (tokens) não é relevante na **análise léxica**!

Papel do Analisador Léxico

- O analisador léxico pode ainda realizar tarefas secundárias:
  - Remoção de espaços
  - Remoção de comentários
  - Exibir mensagens de erro
  - Processamento de macros

Tokens, Padrões e Lexemas

- Usamos os termos "token", "padrão" e "lexema" com significados específicos
- **Token**: unidade léxica
- **Padrão**: regra para reconhecer o token
- **Lexema**: conjunto de caracteres no programa-fonte que é reconhecido pelo padrão de um token

Tokens, Padrões e Lexemas

- Na maioria das linguagens, as seguintes construções são tratadas como *tokens*:
  - Palavras-chave
  - Operadores
  - Identificadores
  - Constantes
  - Literais e cadeias de caracteres
  - Símbolos de pontuação
  - Números

Tokens, Padrões e Lexemas

- Exemplo:

```
printf("Total = %d", score);
```

Token	Lexema	Padrão
id	*printf*	Strings iniciadas com uma letra ou com "_"
par_esq	*(*	"("
string	*"Total = %d"*	Sequência de caracteres entre aspas duplas (" ")
virgula	,	","
id	*score*	Strings iniciadas com uma letra ou com "_"
par_dir	*)*	")"
ponto-virgula	*;*	";"

Observação:
Para a identificação dos tokens pode-se utilizar códigos, ou o próprio símbolo, no caso de palavras reservadas, pontuações, por exemplo.

Atributos para tokens

- Cada token é representado por 3 informações:

  1) **Classe**: identificador, cadeia, etc…

  2) **Lexema** (valor): depende da classe do token. Pode ser o número, uma sequência de caracteres

    - Token simples: não tem valor associado, uma vez que a classe o descreve
    - Token com argumento: tem valor associado

  3) **Posição do token**: local do texto (linha, coluna) onde ocorreu o token. Usado para indicar erros

```
42 + (675 * 31) - 20925
```

Lexema	Tipo	Valor
42	Número	42
+	Operador	SOMA
(	Pontuação	PARESQ
675	Número	675
*	Operador	MULT
31	Número	31
)	Pontuação	PARDIR
–	Operador	SUB
20925	Número	20925

Tabela de símbolos

- Estrutura de dados usada para armazenar as informações sobre os tokens
- Implementação:
  - Listas
  - Árvores
  - Hash
- A estrutura é pesquisada toda vez que um nome é encontrado no código-fonte

Tabela de símbolos

- Operações
  - Inserção
  - Consultar
  - Acessar informações associadas a um nome

- Símbolos pré-definidos, por exemplo, palavras reservadas são inseridas antes de iniciar a compilação

Especificação de tokens

Expressões Regulares

- A especificação de um analisador léxico descreve o conjunto de tokens que formam a linguagem
- A melhor forma de especificarmos os padrões de tokens é por meio de *expressões regulares (ER)*

Especificação de tokens

Expressões Regulares

- Uma expressão regular r é completamente definida pelo conjunto de cadeias de caracteres com as quais ela casa
- Esse conjunto é denominado linguagem gerada pela expressão regular e é denotado como L(r)
- As expressões regulares descrevem todas as linguagens que podem ser formadas a partir de operadores sobre linguagem aplicados aos símbolos de algum alfabeto
- As expressão regulares são construídas recursivamente a partir de expressões regulares menores

Especificação de tokens

Expressões Regulares

- Conceitos básicos:
  - **Símbolo**: Para cada símbolo *a* no alfabeto da linguagem, a expressão regular *a* denota a linguagem contendo apenas a string *a*
  - **Cadeia vazia** - A expressão regular ε representa uma cadeia vazia.
  - **Alternativas** - Se *r* e *s* são expressões regulares, então *r*|*s* é uma expressão regular que casa com qualquer cadeia que case com *r* **ou** com *s*
  - **Concatenação** - A concatenação de duas expressões *r* e *s* é denotada como *rs*, e casa com qualquer cadeia de caracteres que seja a concatenação de duas cadeias, desde que a primeira case com *r* e a segunda com *s*
  - **Repetição** - A operação de repetição é denotada como *r**, onde *r* é uma expressão regular. Isso permite que *r* seja repetida *zero ou mais vezes*

Especificação de tokens

Expressões Regulares

- Há também algumas abreviações possíveis:
  - *[abcd]* é equivalente a *(a|b|c|d)*
  - *[b-g]* é equivalente ao intervalo *[bcdefg]*
  - *[b-gM-Q]* é equivalente ao intervalo *[bcdefgMNOPQ]*
  - *r?* é equivalente a *(r|ε)*, indicando que as cadeias que casam com *r* são opcionais
  - *r+* é equivalente a *(rr*)*, indicando *uma ou mais repetições de r*
  - *~r* é o complemento de *r*

Especificação de tokens

Definições Regulares

- É útil simplificar a notação com nomes para expressões regulares
- Estes nomes podem ser usados como símbolos
- Exemplo:
  - Expressão regular para uma sequência de um ou mais dígitos

    digito digito*

    onde:

    digito = 0 | 1 | 2 | ... | 9

Expressões Regulares para linguagens de programação

Na sequência, são descritas algumas expressões regulares típicas para algumas categorias de tokens

Expressões Regulares para linguagens de programação

- Números: sequências de dígitos (números naturais), números decimais ou números com expoente
- Pode-se escrever expressões regulares para os números como:

  nat = [0-9]+

  nat_sinal = (+|-)? nat +99 -99 99

  decimais = nat_sinal (‘.’ nat)? (‘E’ nat_sinal)?

  9.

Expressões Regulares para linguagens de programação

- Identificadores: Usualmente um identificador deve iniciar com uma letra e conter somente números e dígitos
- Isso pode ser expressado pelas seguintes expressões:

letra = [a-Z]

digito = [0-9]

id = letra(letra|digito)*

Expressões Regulares para linguagens de programação

- Palavras reservadas: Palavra reservadas são mais simples de escrever como expressões regulares

- São representadas por sequencias fixas de caracteres:

  reservadas = if | then | else | while | do | …

Expressões Regulares para linguagens de programação

- Comentários: Comentários são normalmente ignorados durante a varredura, mas para isso devem ser identificados e descartados:

--(~newline)* → *--comentário em Ada*

//(~newline)* → *//comentário em C*

{(~})*} → *{comentário em Pascal}*

Fragmento da linguagem Java

```
&&                          => E_LOGICO
[|][|]                      => OU_LOGICO
[+]                         => ‘+’
[+][+]                      => INC
/                           => ‘/’
[.]                         => ‘.’
while                       => WHILE
if                          => IF
for                         => FOR
else                        => ELSE
[a-zA-Z]                    => ID
[a-zA-Z_][a-zA-Z0-9_]+      => ID
[0-9]+                      => NUM
[0-9]+[.][0-9]+             => NUM
[0-9]+[.]                   => NUM
[.][0-9]+                   => NUM
["]["]                      => STRING
["][^"\n]+["]               => STRING
```

Expressões Regulares para linguagens de programação

Ambiguidade

IMPORTANTE:

Algumas cadeias de caracteres podem casar com diversas expressões regulares.

Exemplos:

- *if* ou *while* podem ser identificadores ou palavras reservadas;
- A cadeia <> poderia representar dois tokens (< e >) ou o símbolo para *diferente*

Há duas regras típicas para os exemplos acima:

(1) Quando há um impasse se o token pode ser identificador ou palavra reservada, opta-se por reconhecer a *palavra reservada*.

(2) Quando uma cadeia pode representar um único token ou uma combinação de dois ou mais tokens, prefere-se identificar a cadeia mais longa. Essa regra é chamada de princípio da *cadeia mais longa*.

Reconhecimento de tokens

AFD's

- Uma outra técnica de representação usada para linguagens regulares são os autômatos finitos
- Os autômatos finitos podem ser utilizados para organizar os padrões léxicos de uma linguagem, facilitando a implementação direta de um analisador léxico para ela
- A transformação de ER's para AFD's está fora do escopo da disciplina
  - Relembrar lá de TC...

Reconhecimento de tokens

AFD's

- Para criar um analisador léxico dessa forma devemos definir os autômatos finitos que representam os padrões associados a cada tipo de token
- Depois, combinamos esses autômatos em um único autômato, e então implementar o autômato finito resultante como um programa
- Um autômato finito possui um conjunto finito de estados;
  - Arestas levam de um estado a outro
  - Cada aresta é rotulada com um símbolo de transição
  - Um estado é o estado inicial
  - Alguns estados são distinguidos como estados finais (que determina que um token foi reconhecido)

Reconhecimento de tokens

AFD's

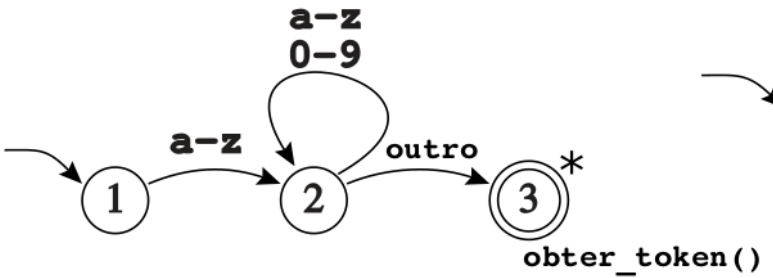
Exemplo:

`if`	`{IF}`
`then`	`{THEN}`
`else`	`{ELSE}`
`[a-z][a-z0-9]*`	`{ID}`
`[0-9]+`	`{NÚMERO INTEIRO}`
`([0-9]+"."[0-9]*)|([0-9]*"."[0-9]+)`	`{NÚMERO REAL}`
`("--"[a-z]*"\n")|(" "|"\n"|"\t")+`	`{COMENTÁRIOS E ESPAÇOS}`

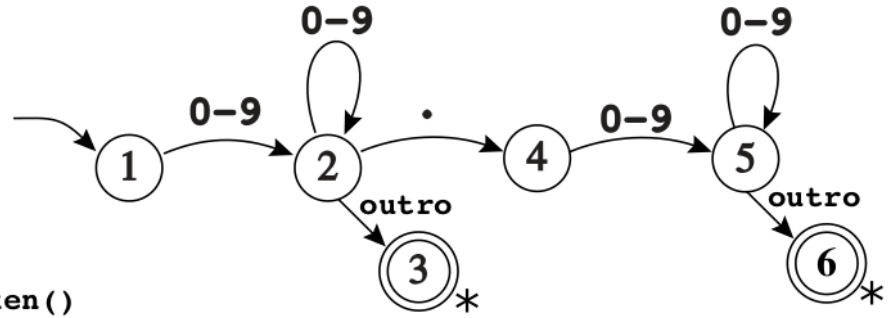
Reconhecimento de tokens

AFD's

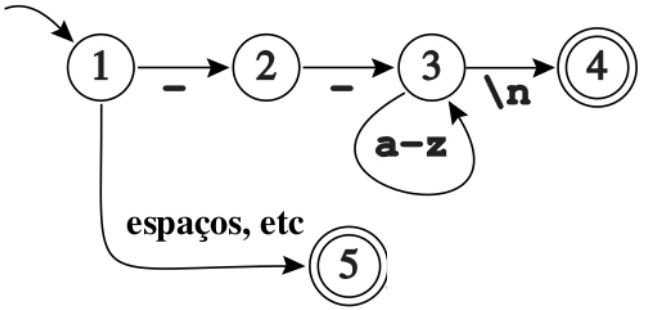
Exemplo:



Identificadores
e palavras-reservadas



Números
(reais e inteiros)

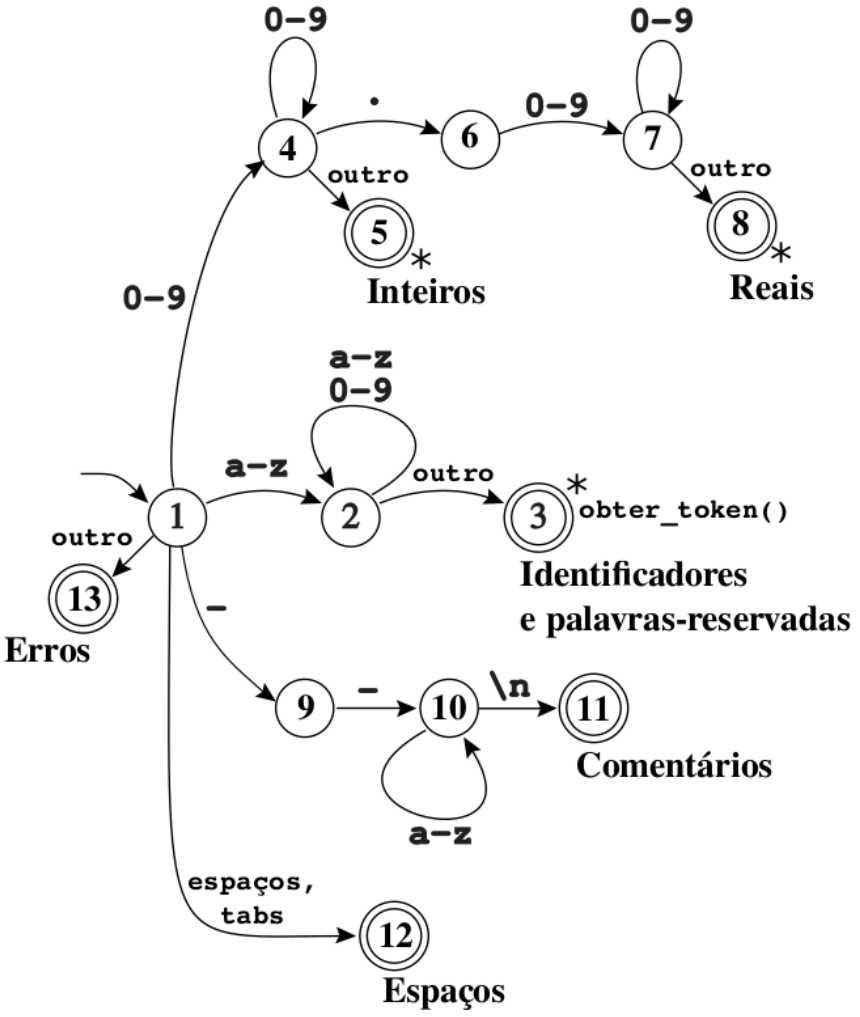


Espaços e
comentários

Um aspecto que chama a atenção nesses autômatos é a presença de transições com o rótulo *outro*.
Isso significa que a transição ocorrerá quando qualquer outro caractere que não esteja especificado nas outras transições for processado.

Reconhecimento de tokens

AFD's



Autômato combinado

(*vulgo automatão*)

Implementando um diagrama de transições

- Uma sequência de diagramas de transições pode ser convertido em um programa que procure pelos tokens especificado pelo diagrama

- Cada estado recebe um segmento de código
  - Se existirem lados deixando um estado, então seu código lê um caractere e seleciona um lado para seguir
  - próximo_caractere( ) → próximo símbolo no buffer
  - Se existir um lado rotulado pelo caracter lido, o controle é transferido para o código apontado pelo lado
  - Se existir, chamar próximo diagrama de transição
  - Se não existir outro diagrama, erro()

Implementando um diagrama de transições

```
estado = 1; //inicio
while (estado=1 ou estado=2)
      case (estado) of
      1: case character_entrada of
         letra: avance_entrada
                estado=2
         else: erro()
         endcase
      2: case (character_entrada) of
         letra: avance_entrada
         digito: avance_entrada
         else: estado=3
         endcase
      endcase
endwhile
```

```
if (estado=3)
   obter_token()
else
   erro()
```

Erros Léxicos

- Uma sequência de caracteres que não pode ser verificada em nenhum token válido resulta em um *erro léxico*
- Na maioria dos casos, um erro léxico é causado pela ocorrência de algum caractere ilegal
- Embora incomuns, esses erros devem ser tratados pelo analisador léxico

Erros Léxicos

Exemplo:

```
fi (a==f(x))
```

- **Que erro é detectado?**

Erros Léxicos

Exemplo:

```
fi (a==f(x))
```

- Não é possível identificar este erro
- "fi" pode ser um identificador de função

Erros Léxicos

- Não é razoável interromper a compilação por causa do que geralmente é um pequeno erro, então geralmente tenta-se algum tipo de recuperação de erro léxico
- Talvez a estratégia mais simples de recuperação de erros seja a da "modalidade pânico"
  - Remove-se sucessivos caracteres da entrada remanescente até que o analisador léxico possa encontrar um token bem formado

Erros Léxicos

- **Exemplo**

  Considere a sequência:

  for$aux

- O símbolo $ terminaria a varredura do token *for*
- Como nenhum token válido começa com *$*, ele será excluído
- Então, a cadeia *aux* seria reconhecida como um identificador

Erros Léxicos

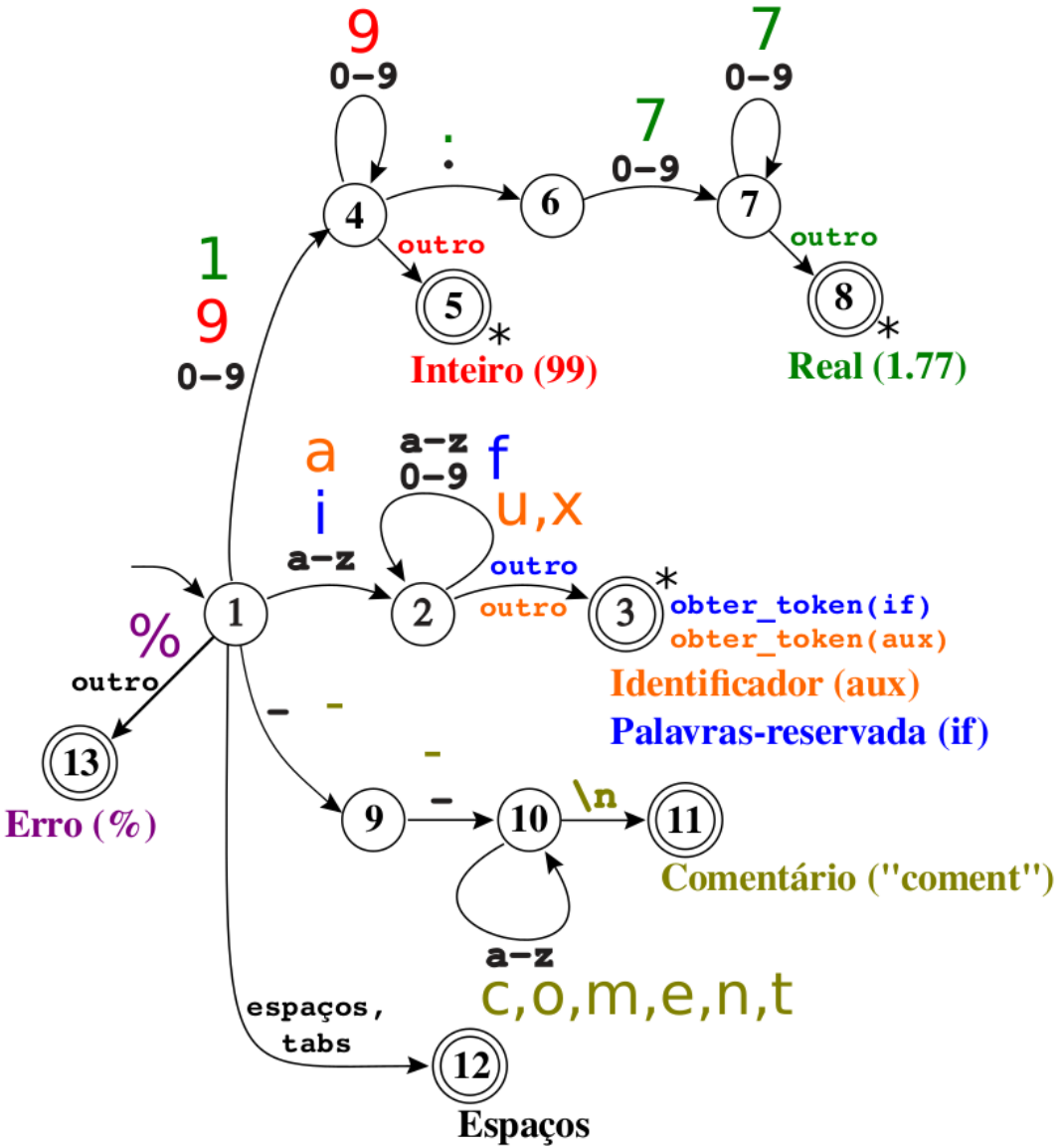
- Outras possibilidades:
    1. remover um caractere estranho;
    2. inserir um caractere ausente;
    3. substituir um caractere incorreto por um correto; ou
    4. transpor dois caracteres adjacentes.

Análise Léxica

Big Example

- Vamos revisitar o autômato (automatão), mas desta vez utilizando-o para realizar a análise léxica de um trecho de código:

```
99 1.77 if aux
--coment
%
```



Análise Léxica:

Considerações Importantes

- Certas convenções de linguagem aumentam a dificuldade da análise léxica
- Um exemplo popular que ilustra a dificuldade potencial em se reconhecer tokens é o enunciado DO de Fortran, uma vez que a linguagem *não considera espaços em branco*
- No comando:

  DO 5 I = 1.25

  não podemos afirmar que DO seja parte do identificador DO5I, e não um identificador em si, até que tenhamos examinado o ponto decimal

  - Nesse caso teríamos uma atribuição do valor *1.25* para a variável *DO5I*

Análise Léxica:

Considerações Importantes

- Por outro lado, no enunciado:

  DO 5 I = 1,25

  temos sete tokens:

  1) palavra-chave DO
  2) rótulo de enunciado 5
  3) identificador I
  4) operador =
  5) constante 1
  6) vírgula
  7) constante 25

- Aqui não podemos estar certos, até que tenhamos examinado a vírgula, de que DO seja uma palavra-chave (laço de repetição)

Análise Léxica:

Considerações Importantes

- Em PL/I, as palavras-chave não são reservadas
- Consequentemente, as regras para essa distinção são um tanto complicadas, como o seguinte enunciado PL/I ilustra:

```
IF THEN THEN THEN = ELSE;
ELSE ELSE = THEN;
```

- Nesse caso, temos os seguintes tokens (não muito fáceis de identificar):

<IF, >

<id, THEN>

<THEN, >

<id, THEN>

<=, >

<id, ELSE>

<; , >

<ELSE, >

<id, ELSE>

<=, >

<id, THEN>

<; , >

Análise Léxica:

Considerações Importantes

- Na linguagem Python, usa-se a indentação para especificar sentenças compostas. No código:

```
if x > y :
   x=y
   print "case 1"
```

todas as sentenças indentadas igualmente são incluídas no mesmo bloco

- Nesse caso, o analisador léxico não pode descartar os espaços em branco no início das linhas, pois eles são considerados na sintaxe do programa

Análise Léxica:

Considerações Importantes

- Para a delimitação de blocos de códigos os delimitadores são colocados em uma pilha e diferenciados por INDENT e DEDENT

```
            def perm(l):                          NOVA LINHA
    INDENT      if len(l) <= 1:                   NOVA LINHA
    INDENT        return[1]                       NOVA LINHA
    DEDENT      r = [ ]                           NOVA LINHA
                for i in range(len(l)):           NOVA LINHA
    INDENT          s = l[:i] + l[i+1:]           NOVA LINHA
                    p = perm(s)                   NOVA LINHA
                    for x in p:                   NOVA LINHA
    INDENT              r.append(l[i:i+1]+x)      NOVA LINHA
    DEDENT      return r
```

https://docs.python.org/2/reference/lexical_analysis.html

Análise Léxica:

Resumo

- **O que?**
  - Separação de tokens
  - Classificar os tokens
  - Sinalização de erros
- **Quando?**
  - Início do front-end
- **Como?**
  - Expressões Regulares
  - Autômatos

- **Entrada:**
  - String (Código-fonte)
- **Saída:**
  - Sequência de tokens (entrada do An. Sintático)
  - Mensagens Erros

Análise Léxica:

Leitura Recomendada

- **Apostila: Capítulo 2**
- **Livro Dragão: Capítulo 3**

- **Vídeos:**

  https://www.youtube.com/watch?v=kazE4zPRsxc

  https://www.youtube.com/watch?v=VGgIZl5WjHO&feature=youtu.be

Análise Léxica:

Exercícios

- **Apostila: Capítulo 2**
  - 1-5

- **Livro Dragão (1ª Ed.): Capítulo 3**
  - 3.2
  - 3.3
  - 3.7