Análise Léxica

Alexsandro Santos Soares

Universidade Federal de Uberlândia
Faculdade de Computação

Introdução

- Um analisador léxico recebe uma sequência de caracteres e produz uma sequência de nomes, palavras-chaves e sinais de pontuação, chamados de tokens.
- Ele descarta espaços em branco e comentários entre os tokens.
- Ao realizar a sua tarefa o analisador léxico simplifica a construção do analisador sintático.
  - Essa é a razão principal para separar as duas etapas.

Tokens lexicais

- Um token lexical é uma sequência de caracteres que pode ser tratada como uma unidade da gramática de uma linguagem de programação.
- Uma linguagem de programação classifica os tokens em um conjunto finito de tipos.

Exemplos de Tokens

Tipo	Exemplos
ID	`fab n14 ultimo`
NUM	`73 0 00 515 082`
REAL	`66.1 .5 10. 1e67 5.5e-10`
IF	`if`
VIRG	`,`
DIF	`!=`
APAR	`(`
FPAR	`)`

Tokens lexicais

- Em muitas linguagens, tokens tais como IF, VOID, RETURN são chamados de palavras reservadas e não podem ser usados como identificadores.

Exemplos de Não tokens

comentário	`/* comentário */`
diretivas do pré-processador	`#include <stdio.h>`
diretivas do pré-processador	`#define NUMS 5 , 6`
macro	`NUMS`
brancos, tabs e novas linhas	

Exemplo de saída

Para o programa a seguir

```
float encontra0(char *s) /* encontra um zero */
{
   if (!strncmp(s, "0.0", 3))
      return 0.0;
}
```

O analisador léxico poderia retornar a sequência

```
FLOAT  ID(encontra0) APAR CHAR ASTER ID(s) FPAR
ACHAVE IF APAR EXCLA ID(strncmp) APAR ID(s) VIRG
STRING(0.0) VIRG NUM(3) FPAR FPAR RETURN REAL(0.0)
PTVIRG FCHAVE EOF
```

Alguns tokens, tais como identificadores e literais, possuem valores semânticos associados a eles, fornecendo informação auxiliar além do tipo.

Especificação de tokens

- Como as regras lexicais de uma linguagem de programação deveriam ser descritas?
- Poderíamos especificar tokens lexicais em português. Abaixo está uma descrição dos identificadores em C ou Java:

Definição 1

Um identificador é uma sequência de letras e dígitos; o primeiro caracter deve ser uma letra. O sublinhado _ conta como uma letra. Letras maiúsculas e minúsculas são diferentes. Se o fluxo de entrada foi analisado até um dado caracter, o próximo token incluirá a maior cadeia de caracteres que poderia constituir um token. Brancos, tabs, novas linhas e comentários são ignorados exceto quando servirem para separar tokens. Pelo menos um espaço em branco é necessário para separar identificadores, palavras-chaves e constantes adjacentes.

- Especificaremos os tokens usando a linguagem formal das expressões regulares e implementaremos os analisadores lexicais usando autômatos finitos determinísticos.
  - Isso levará a analisadores mais simples e mais legíveis.

Expressões regulares

- Vamos definir linguagem como um conjunto de strings.
- Uma string é uma sequência finita de símbolos.
- Os símbolos são retirados de um alfabeto finito.
- Como exemplos:
  - A linguagem Pascal é o conjunto de todas as strings que constituem programas legais em Pascal.
  - A linguagem das palavras reservadas em C é o conjunto de todas as strings alfabéticas que não podem ser usads como identificadores em C.
- Para especificar algumas destas linguagens, possivelmente infinitas, com descrições finitas, usaremos a notação das *expressões regulares*.

Definição 2

Uma expressão regular (ER) compreende um conjunto de strings formadas da seguinte forma:

Símbolo para cada símbolo **a** no alfabeto da linguagem, a ER **a** denota a linguagem contendo apenas a string `a`.

Alternativa dadas duas ERs M e N, o operador de alternativa escrito com uma barra vertical | cria uma nova expressão regular $M|N$. Uma string está na linguagem de $M|N$ se ela estiver na linguagem de M ou na de N. Assim, a linguagem $\mathbf{a}|\mathbf{b}$ contém duas strings `a` e `b`.

Concatenação dadas duas ERs M e N, o operador de concatenação $\cdot$ cria uma nova expressão regular $M \cdot N$. Uma string está na linguagem de $M \cdot N$ se ela for a concatenação de quaisquer duas strings α e β, tal que α está na linguagem de M e β na de N. Assim, a ER $(\mathbf{a}|\mathbf{b}) \cdot \mathbf{a}$ define a linguagem contendo as strings `aa` e `bb`.

Epsilon A expressão regular ϵ representa a linguagem cuja única string é a string vazia. Assim, $(\mathbf{a} \cdot \mathbf{b})|\epsilon$ representa a linguagem {`""`, `"ab"`}.

Repetição dadas uma ER M, seu fechamento de Kleene é M^*. Uma string está em M^* se ela for a concatenação de zero ou mais strings de M. Assim, $((\mathbf{a}|\mathbf{b}) \cdot \mathbf{a})*$ representa o conjunto infinito {`""`, `"aa"`, `"ba"`, `"aaaa"`, `"baaa"`, ... }.

Especificação de tokens

Exemplos simples

$(\mathbf{0}\|\mathbf{1})^* \cdot \mathbf{0}$	Números binários múltiplos de dois.
$\mathbf{b}^*(\mathbf{abb}^*)^*(\mathbf{a}\|\epsilon)$	Strings de `a`'s e `b`'s sem `a`'s consecutivos
$(\mathbf{a}\|\mathbf{b})^*\mathbf{aa}(\mathbf{a}\|\mathbf{b})^*$	Strings de `a`'s e `b`'s com `a`'s consecutivos

- Usando expressões regulares podemos especificar o conjunto de caracteres ASCII correspondente aos tokens lexicais de uma linguagem de programação.
- Ao escrever uma ER faremos as seguinte abreviações
  - Os símbolos de concatenação ou epsilon serão omitidos e o fechamento de Kleene possui maior precedência que os demais operadores. Assim, $\mathbf{ab}|\mathbf{c}$ significa $\mathbf{a} \cdot \mathbf{b}|\mathbf{c}$ e $(\mathbf{a}|)$ significa $(\mathbf{a}|\epsilon)$.
  - $[\mathbf{abcd}]$ significa $(\mathbf{a}|\mathbf{b}|\mathbf{c}|\mathbf{d})$; $[\mathbf{b}-\mathbf{g}]$ significa $[\mathbf{bcdefg}]$ e $[\mathbf{b}-\mathbf{gM}-\mathbf{Qkr}]$ significa $[\mathbf{bcdefgMNOPQkr}]$.
  - $M?$ significa $(M|\epsilon)$ e M^+ significa $(M \cdot M^*)$.
  - `"a.+*"` qualquer coisa entre parênteses denota a própria string.
  - `.` significa qualquer caracter que não seja o nova linha.

Especificação de tokens

Exemplos de tokens

`if`	IF
`[a-z][a-z0-9`	ID
`[0-9]+`	NUM
`([0-9]+"."[0-9]*)|([0-9]+"."[0-9]+)`	REAL
`("//"[a-z]*"\n")|(" "|"\n"|"\t")+`	*não faz nada*
`.`	`erro()`

- A quinta linha da descrição reconhece comentários e espaços em branco, mas os descarta.
- Uma especificação lexical deve ser completa, sempre casando com alguma substring inicial da entrada. Isto pode ser feito com uma regra que case com qualquer caracter, como na última linha.

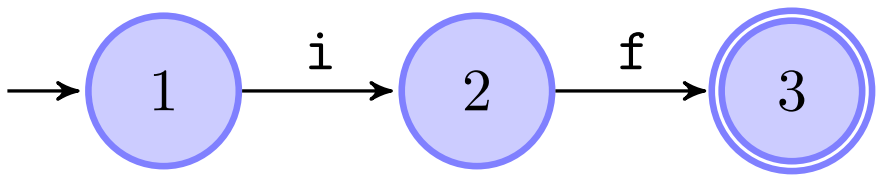
Ambiguidades na especificação

- O que acontece se encontrarmos `if8` na entrada? Ela deve ser reconhecida como um identificador ou como dois tokens `if` e `8`?
- A entrada `if 89` começa com um identificador ou com uma palavra reservada?
- Para resolver estas questões existem duas regras de desambiguidade normalmente usadas em geradores de analisadores lexicais:

  Casamento mais longo a maior substring inicial da entrada que puder casar com qualquer expressão regular será tomada como o próximo token.

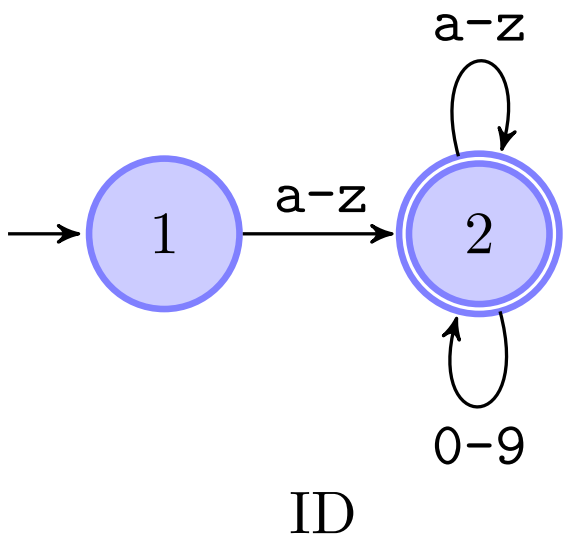
  Prioridade da regra Para uma substring inicial *particular*, a primeira expressão regular que casar com ela determina o tipo do token. Isso significa que a ordem de escrita das regras importa.
- Assim `if8` será um identificador pela regra do casamento mais longo e `if` será casada como uma palavra reservada pela regra da prioridade.

Autômato finito para if

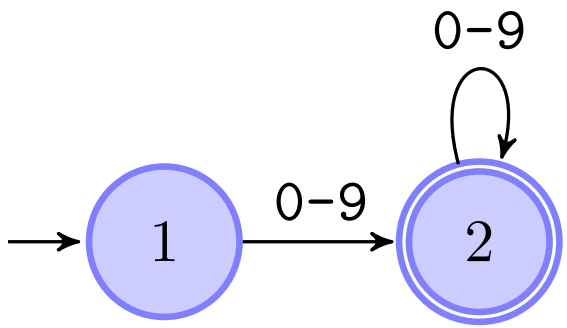


IF

Autômato finito para identificadores

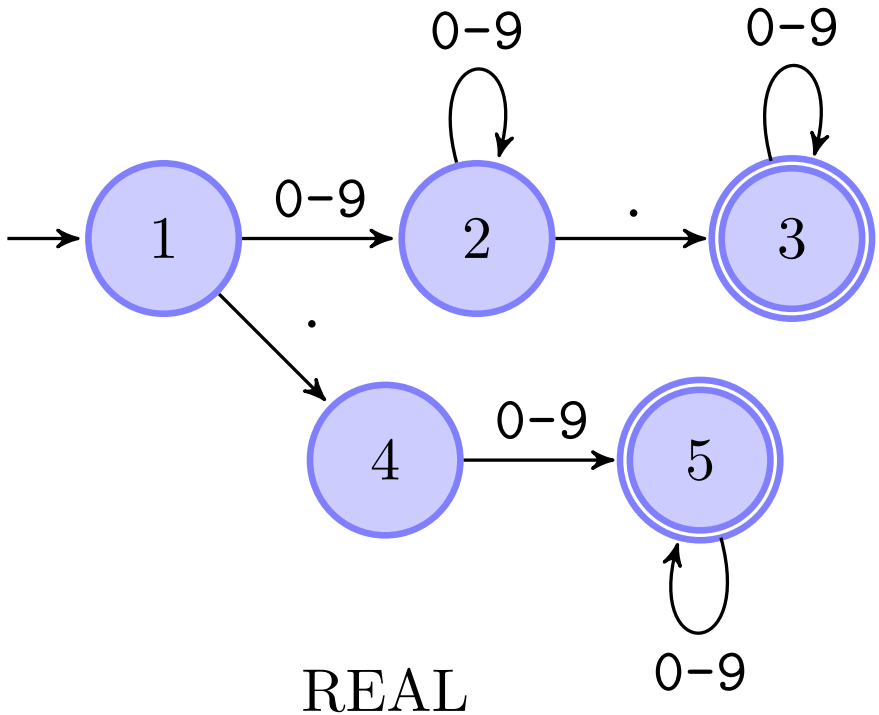


Autômato finito para números inteiros

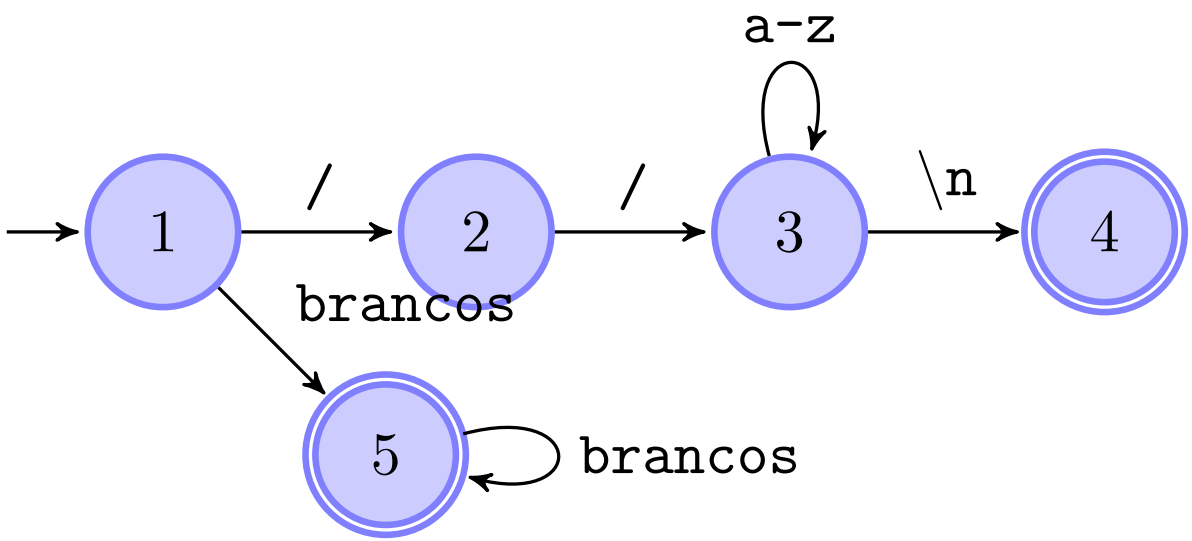


NUM

Autômato finito para números reais

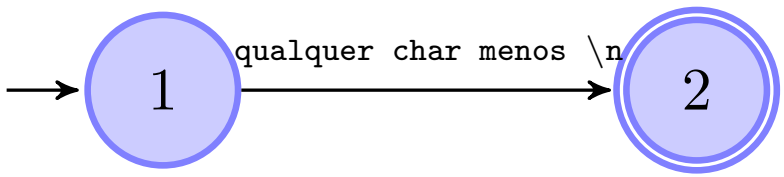


Autômato finito para espaços em branco



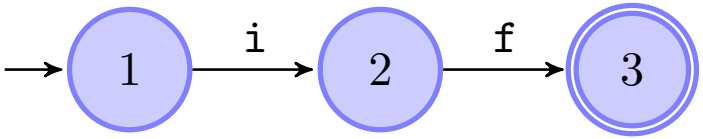
espaço em branco

Autômato finito para um caracter desconhecido

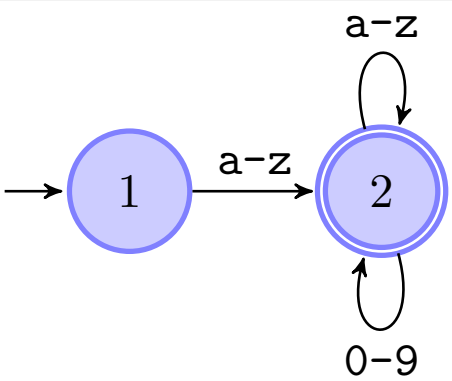


erro

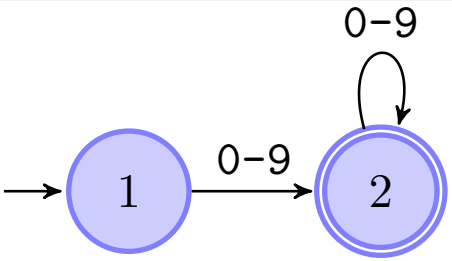
Como combinar os vários autômatos?



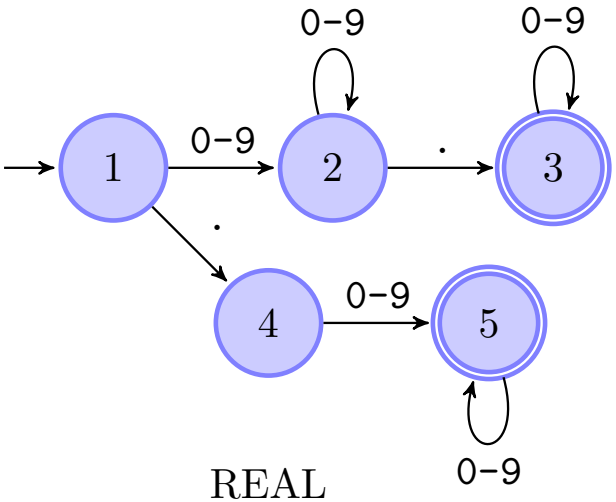
IF



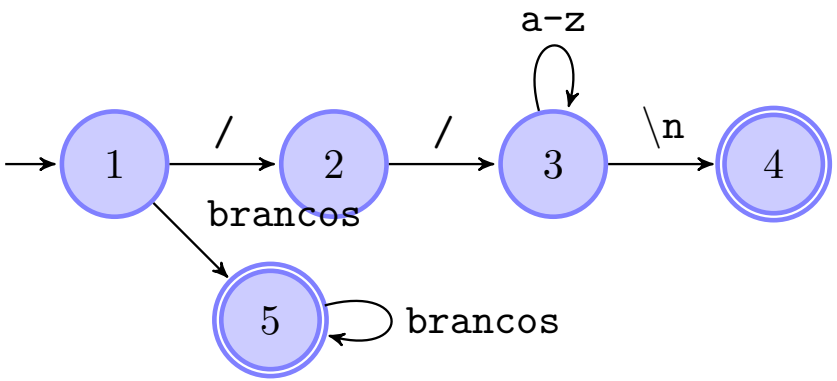
ID



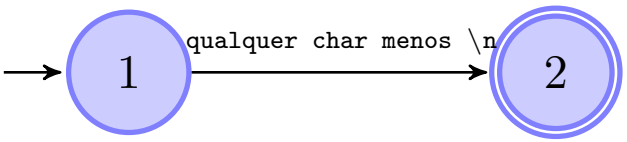
NUM



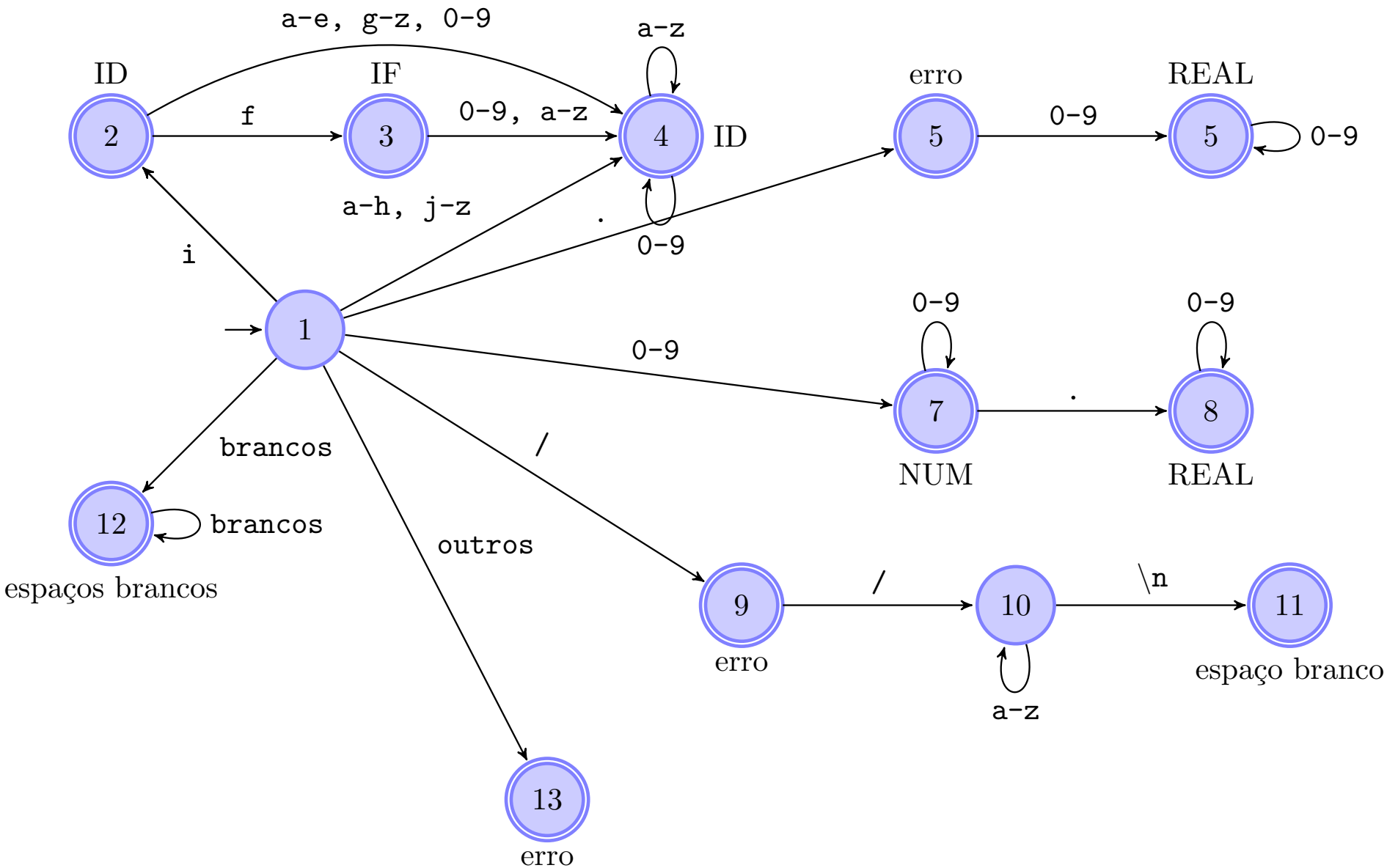
REAL



espaço em branco



Autômato finito combinado



Ocamllex: um gerador de analisadores lexicais

- Ocamllex é um gerador de analisadores lexicais que produz um programa Ocaml a partir de uma *especificação lexical*.
- Para cada tipo de token na linguagem de programação a ser analisada lexicamente, a especificação conterá uma expressão regular e uma *ação*.
- A ação comunica o tipo do token, juntamente com outras informações, para a próxima fase do compilador.
  - As ações são trechos de expressões Ocaml que retornam valores para os tokens.
- O arquivo contendo a especificação tem extensão `.mll` e o arquivo de saída, contendo o código Ocaml do analisador, possui extensão `.ml`
- A chamada típica da ferramenta é

```
ocamllex lexico.mll
```

Formato geral do arquivo de entrada

O arquivo de entrada do ocamllex possui quatro seções: cabeçalho, definições, regras e rodapé.

```
(* Cabeçalho *)
{ cabeçalho }

(* Definições *)
let ident = regexp
let ...

(* Regras *)
rule ponto_de_entrada [arg1, ..., argn] = parse
  | padrão { ação }
  | ...
  | padrão { ação }
and ponto_de_entrada2 [arg1, ..., argk] = parse
 ...
and ...

(* Rodapé *)
{ rodapé }
```

As seções do arquivo de entrada

- As seções de definições e de rodapé são *opcionais*.
- O cabeçalho e o rodapé contém código Ocaml arbitrário que serão copiados, *ipsis litteris*, respectivamente para o início e para o final do arquivo `.ml` gerado.
  - Normalmente, colocamos diretivas de abertura de módulos e algumas funções auxiliares no cabeçalho.
- As definições contém declarações de definições (*abreviações*) que simplificarão a especificação das expressões regulares posteriores. Por exemplo:

```
let digito = ['0' - '9']
let inteiro = digito+

let letra = ['a' - 'z' 'A' - 'Z']
```

As seções do arquivo de entrada

- A seção regras contém uma função lexical por ponto de entrada
  - O nome da função é o mesmo que o usado como `ponto_de_entrada`.
  - Cada ponto de entrada se tornará uma função Ocaml que receberá [n+1] argumentos, sendo o último deles implícito e do tipo `Lexing.lexbuf`
  - Os argumentos *arg1, ..., argn* são para uso nas ações associadas aos padrões.

```
rule ponto_de_entrada [arg1, ..., argn] = parse
  | padrão { ação }
  | ...
  | padrão { ação }
and ...
```

Padrões

Os padrões contém as expressões regulares que definem um token:

'c'	casa com o caracter delimitado pelos apóstrofos.
_	o sublinhado casa com qualquer caracter.
eof	casa como o fim de arquivo.
"string"	casa com o literal string delimitado pelas aspas.
$e_1 \mid e_2$	significa uma escolha ou alternativa.
$[c_1 - c_2]$	escolha de qualquer caracter no intervalo fechado entre c_1 e c_2, como determinado pela tabela ASCII.
$[\hat{} c_1 - c_2]$	escolha de qualquer caracter que não esteja no intervalo fechado entre c_1 e c_2.
$r*$	casa com zero ou mais expressões regulares `r`.
$r+$	casa com uma ou mais expressões regulares `r`.
$r?$	casa com zero ou uma expressão regular `r`.

Padrões

ident uma das abreviações definidas anteriormente na seção de definições.

e_1 as id liga o resultado do casamento de e_1 com `id` para uso na *ação*.

Exemplo de arquivo .mll

- Vamos usar o ocamllex para gerar um analisador léxico para uma linguagem especificada da seguinte forma:
  - As únicas palavras reservadas são `if` e `while`.
  - O comando de atribuição é sinalizado por `:=`
  - O único operador é `+` e pode operar sobre números inteiros e identificadores. Parênteses também podem ser usados.
  - Um identificador começa com letra mas pode ser seguido por qualquer combinação de letras, dígitos ou sublinhados.
  - A linguagem possui comentários de linha iniciados por `//`.
  - A linguagem também possui strings, delimitadas por `"` e comentários de múltiplas linhas que podem ser aninhados e são delimitados por `/*` e `*/`.
- Para simplificar a exposição não incluirei na primeira versão as strings e os comentários aninhados.

Especificação inicial – cabeçalho

```
{
  open Lexing
  open Printf

  type tokens = APAR
              | FPAR
              | ATRIB
              | IF
              | WHILE
              | MAIS
              | LITINT of int
              | LITSTRING of string
              | ID of string
              | EOF

  let incr_num_linha lexbuf =
    let pos = lexbuf.lex_curr_p in
     lexbuf.lex_curr_p <- { pos with
        pos_lnum = pos.pos_lnum + 1;
        pos_bol = pos.pos_cnum;
     }
```

Especificação inicial – continuação do cabeçalho

```
let msg_erro lexbuf c =
    let pos = lexbuf.lex_curr_p in
    let lin = pos.pos_lnum
    and col = pos.pos_cnum - pos.pos_bol - 1 in
    sprintf "%d-%d: caracter desconhecido %c" lin col c

  let erro lin col msg =
    let mensagem = sprintf "%d-%d: %s" lin col msg in
       failwith mensagem

}
```

Lembre-se que o cabeçalho é delimitado por `{}`.

Especificação inicial – definições de abreviações

```
let digito = ['0' - '9']
let inteiro = digito+

let letra = ['a' - 'z' 'A' - 'Z']
let identificador = letra ( letra | digito | '_')*

let brancos = [' ' '\t']+
let novalinha = '\r' | '\n' | "\r\n"

let comentario = "//" [^ '\r' '\n' ]*
```

Especificação inicial – regras lexicais

```
rule token = parse
  brancos    { token lexbuf }
| novalinha { incr_num_linha lexbuf; token lexbuf }
| comentario { token lexbuf }
| '('        { APAR }
| '+'        { MAIS }
| ')'        { FPAR }
| ":="       { ATRIB }
| inteiro as num { let numero = int_of_string num in
                   LITINT numero }
| "if"       { IF }
| "while"    { WHILE }
| identificador as id { ID id }
| _ as c     { failwith (msg_erro lexbuf c) }
| eof        { EOF }
```

Note que as abreviações foram usadas aqui.

Gerando o analisador léxico

- Para gerar o analisador lexico a partir do arquivo `lexico.mll` contendo a especificação:

```
ocamllex lexico.mll
ocamlc -c lexico.ml
```

A última linha serve para compilar o arquivo `lexico.ml` gerado.

- Para usar o analisador léxico, entre no Ocaml

```
rlwrap ocaml
```

e dentro do intérprete digite:

```
#load "lexico.cmo";;
```

Isso serve para carregar o arquivo compilado. Antes de realmente vermos os tokens analisados, precisamos informar de onde virá o fluxo de caracteres (inicialmente, de uma string):

```
let lexbuf = Lexing.from_string "if while 123 // comentario";;
```

Usando o analisador léxico

- Agora, para vermos os tokens analisados, fazemos:

```
# Lexico.token lexbuf;;
- : Lexico.tokens = Lexico.IF

# Lexico.token lexbuf;;
- : Lexico.tokens = Lexico.WHILE

# Lexico.token lexbuf;;
- : Lexico.tokens = Lexico.LITINT 123

# Lexico.token lexbuf;;
- : Lexico.tokens = Lexico.EOF
```

Note que o último token é o de fim de arquivo e é gerado quando a string terminar de ser analisada.

Módulo de carregamento

Ao invés de usar a linha de comando do intérprete para gerar os tokens um a um, podemos escrever um módulo que gere todos os tokens em uma lista. Chamarei este módulo de `carregador.ml`:

```
#load "lexico.cmo";;

let rec tokens lexbuf =
  let tok = Lexico.token lexbuf in
  match tok with
  | Lexico.EOF -> [Lexico.EOF]
  | _ -> tok :: tokens lexbuf

let lexico str =
  let lexbuf = Lexing.from_string str in
  tokens lexbuf

let lex arq =
  let ic = open_in arq in
  let lexbuf = Lexing.from_channel ic in
  let toks = tokens lexbuf in
  let _ = close_in ic in toks
```

Módulo de carregamento

O procedimento para gerar e compilar o analisador léxico continua o mesmo

```
ocamllex lexico.mll
ocamlc -c lexico.ml
```

Depois entramos no intérprete do Ocaml

```
rlwrap ocaml
```

E carregamos o módulo `carregador.ml`:

```
#use "carregador.ml";;
```

Testando o analisador

Ao invés do fluxo de caracteres vir de uma string, queremos que ele venha de arquivo de nome `teste.txt`:

```
if while 123 // comentario
linha2
while2
```

Com esse arquivo, digitamos no console do intérprete

```
lex "teste.txt";;

- : Lexico.tokens list =
[Lexico.IF; Lexico.WHILE; Lexico.LITINT 123; Lexico.ID "linha2";
 Lexico.ID "while2"; Lexico.EOF]
```

Especificação completa do analisador léxico

- Agora, vamos incluir na especificação as strings e os comentários de múltiplas linhas, que podem ser aninhados.
- As seções de cabeçalho e de definições não serão modificadas.
- Apenas acrescentaremos algumas novas regras na especificação e dois novos pontos de entrada: um para comentários e outro para strings.
- No próximo slide será mostrado o primero ponto de entrada, já modificado.

```
rule token = parse
  brancos    { token lexbuf }
| novalinha { incr_num_linha lexbuf; token lexbuf }
| comentario { token lexbuf }
| "/*"       { comentario_bloco 0 lexbuf }
| '('        { APAR }
| '+'        { MAIS }
| ')'        { FPAR }
| ":="       { ATRIB }
| inteiro as num { let numero = int_of_string num in LITINT numero }
| "if"       { IF }
| "while"    { WHILE }
| identificador as id { ID id }
| '"'        { let pos = lexbuf.lex_curr_p in
               let lin = pos.pos_lnum
               and col = pos.pos_cnum - pos.pos_bol - 1 in
               let buffer = Buffer.create 1 in
               let str = leia_string lin col buffer lexbuf in
                 LITSTRING str }
| _ as c { failwith (msg_erro lexbuf c) }
| eof        { EOF }
```

Autômato para strings

```
and leia_string lin col buffer = parse
   '"'     { Buffer.contents buffer}
| "\\t"    { Buffer.add_char buffer '\t';
            leia_string lin col buffer lexbuf }
| "\\n"    { Buffer.add_char buffer '\n';
            leia_string lin col buffer lexbuf }
| '\\' '"' { Buffer.add_char buffer '"';
             leia_string lin col buffer lexbuf }
| '\\' '\\' { Buffer.add_char buffer '\\';
             leia_string lin col buffer lexbuf }
| _ as c    { Buffer.add_char buffer c;
             leia_string lin col buffer lexbuf }
| eof       { erro lin col "A string não foi fechada"}
```

Autômato para comentários aninhados

Agora introduziremos as regras para tratar comentários aninhados

```
and comentario_bloco n = parse
   "*/"      { if n=0 then token lexbuf
               else comentario_bloco (n-1) lexbuf }
| "/*"       { comentario_bloco (n+1) lexbuf }
| novalinha { incr_num_linha lexbuf;
               comentario_bloco n lexbuf }
| _          { comentario_bloco n lexbuf }
| eof        { failwith "Comentário não fechado" }
```

Uso da nova especificação

Para usar a nova especificação, vou redefinir o arquivo `testes.txt`:

```
if while 123 // comentario
linha2 "abc\"def" /*
  comentario 1
*/
z := "abc" + "def"
while2
```

O procedimento para a geração e uso do analisador é o mesmo:

```
ocamllex lexico.mll
ocamlc -c lexico.ml
```

Depois entramos no intérprete do Ocaml

```
rlwrap ocaml
```

E carregamos o módulo `carregador.ml`:

```
#use "carregador.ml";;
```

Blocos de código e indentação

- Linguagens como Python ou Haskell usam identação para marcar o início e o final de um bloco de comandos.
  - Evita o uso de marcadores tais como chaves ou *begin* – *end*.
  - Evita o uso de terminadores de comandos tais como ;
- Somente os espaços brancos iniciais importam. Os demais podem ser ignorados como em outras linguagens.

```
1 def fat(n):
2     """ Calcula o fatorial de n."""
3     print('n =', n)
4     if n > 1:
5         return n * fat(n - 1)
6     else:
7         print('fim da linha')
8         return 1
```

Indentação em Python

- A primeira tarefa do analisador léxico para Python é substituir as tabulações, da esquerda para a direita, por um a oito espaços tal que o número total de caracteres desde o início da linha, incluindo o substituido, é um múltiplo de oito.
  - Mesma regra usada pelo Unix
- O número total de espaços precedendo o primeiro caractere não branco determina a indentação da linha.
- Os níveis de indentação de linhas consecutivas são usadas para gerar os tokens INDENTA e DEDENTA, usando uma pilha.

Geração de tokens INDENTA e DEDENTA

- Antes que a primeira linha do arquivo seja lida, um zero é empilhado. Ele nunca será desempilhado.
- Os números empilhados serão sempre estritamente crescentes da base para o topo.
- No início de cada linha lógica, o nível de indentação da linha é comparado com o topo da pilha:
  - Se for igual, nada aconteçe.
  - Se for maior, ele é empilhado e um token INDENTA é gerado.
  - Se for menor, ele deve ser igual a um dos números já empilhados. Todos os números empilhados que forem maiores que o nível de indentação serão desempilhados e um token DEDENTA é gerado para cada um.
- No final do arquivo, um token DEDENTA é gerado para cada número ainda na pilha que for maior que zero.

Exemplo de código mal indentado

Abaixo está um exemplo de código contendo vários erros de indentação:

```
1      def perm(l):                        # erro: primeira linha indentada
2     for i in range(len(l)):              # erro: não indentado
3         s = l[:i] + l[i+1:]
4             p = perm(l[:i] + l[i+1:]) # erro: indentação não esperada
5             for x in p:
6                     r.append(l[i:i+1] + x)
7                 return r                # erro: dedentação inconsistente
```

- Os primeiros três erros serão detectados pelo analisador sintático.
- O último erro é encontrado pelo analisador léxico.
  - A indentação de `return r` não se alinha com qualquer nível presente na pilha.

Pré processador para o Python

```
open Lexico; open Printf
let preprocessa lexbuf =
  let pilha = Stack.create ()
  and npar = ref 0 in
    let _ = Stack.push 0 pilha in
    let off_side toks nivel =
    if !npar != 0 (* nova linha entre parênteses *)
    then toks     (* não faça nada *)
    else if nivel > Stack.top pilha
         then begin
           Stack.push nivel pilha; INDENTA :: toks
         end
        else if nivel = Stack.top pilha then toks
        else begin
          let prefixo = ref toks in
          while nivel < Stack.top pilha do
            ignore (Stack.pop pilha);
            if nivel > Stack.top pilha then failwith "Erro"
            else prefixo := DEDENTA :: !prefixo
          done;
          !prefixo
        end
```

Pré processador para o Python – continuação

```
let rec dedenta sufixo =
  if Stack.top pilha != 0
  then let _ = Stack.pop pilha in
       dedenta (DEDENTA :: sufixo)
  else sufixo
in
let rec get_tokens () =
  let tok = Lexico.preprocessador 0 lexbuf in
  match tok with
    Linha(nivel,npars,toks) ->
    let new_toks = off_side toks nivel in
    npar := npars;
    new_toks @ (if npars = 0
                then NOVALINHA :: get_tokens ()
                else get_tokens ())
   | _ -> dedenta []
in get_tokens ()
```

Pré processador para o Python – continuação

```
(* Chama o analisador léxico *)
let lexico =
  let tokbuf = ref None in
  let carrega lexbuf =
    let toks = preprocessa lexbuf in
    (match toks with
       tok::toks -> tokbuf := Some toks; tok
     | []        -> print_endline "EOF"; EOF)
  in
  fun lexbuf ->
    match !tokbuf with
    | Some tokens ->
       (match tokens with
        | tok::toks -> tokbuf := Some toks; tok
        | [] -> carrega lexbuf)
    | None -> carrega lexbuf
```

Mini analisador léxico para o Python : lexico.mll

```
{
  open Lexing
  open Printf

  type token =
  | LITINT of (int)
  | LITSTRING of (string)
  | ID of (string)
  | APAR | FPAR
  | VIRG | MAIS
  | DPONTOS
  | MENORIGUAL
  | SETA | E
  | ATRIB | RETURN
  | DEF
  | EOF
  (* Os tokens a seguir são importantes para o pré processador *)
  | Linha of (int * int * token list)
  | INDENTA
  | DEDENTA
  | NOVALINHA
```

Mini analisador léxico para o Python : lexico.mll

```
  let nivel_par = ref 0

  let incr_num_linha lexbuf =
    let pos = lexbuf.lex_curr_p in
      lexbuf.lex_curr_p <- { pos with
         pos_lnum = pos.pos_lnum + 1;
         pos_bol = pos.pos_cnum;
      }

  let msg_erro lexbuf c =
    let pos = lexbuf.lex_curr_p in
    let lin = pos.pos_lnum
    and col = pos.pos_cnum - pos.pos_bol - 1 in
    sprintf "%d-%d: caracter desconhecido %c" lin col c

}
```

Mini analisador léxico para o Python : lexico.mll

```
let digito = ['0' - '9']
let int = digito+
let comentario = "#"[ ^ '\n' ]*
let linha_em_branco = [' ' '\t' ]* comentario
let restante = [^ ' ' '\t' '\n' ] [^ '\n']+
let brancos = [' ' '\t']+
let novalinha = '\r' | '\n' | "\r\n"

let letra = [ 'a'-'z' 'A' - 'Z']
let identificador = letra ( letra | digito | '_' )*
```

Mini analisador léxico para o Python : lexico.mll

```
(* O pré processador necessário para contabilizar a indentação *)
rule preprocessador indentacao = parse
  linha_em_branco       { preprocessador 0 lexbuf } (* ignora *)
| [' ' '\t' ]+ '\n'     { incr_num_linha lexbuf;
                          preprocessador 0 lexbuf } (* ignora *)
| ' '                   { preprocessador (indentacao + 1) lexbuf }
| '\t'                  { let nova_ind =
                               indentacao + 8 - (indentacao mod 8)
                          in preprocessador nova_ind lexbuf }
| novalinha             { incr_num_linha lexbuf;
                          preprocessador 0 lexbuf }
| restante as linha {
      let rec tokenize lexbuf =
          let tok = token lexbuf in
            match tok with
            | EOF -> []
            | _ -> tok :: tokenize lexbuf
          in let toks = tokenize (Lexing.from_string linha)
             in Linha(indentacao,!nivel_par, toks)
}
| eof { nivel_par := 0; EOF }
```

Mini analisador léxico para o Python : lexico.mll

```
(* O analisador léxico a ser chamado após o pré processador *)
and token = parse
  brancos           { token lexbuf }
| comentario        { token lexbuf }
| "/*"              { comentario_bloco 0 lexbuf; }
| '('               { let _ = incr(nivel_par) in APAR }
| ')'               { let _ = decr(nivel_par) in FPAR }
| ','               { VIRG }
| '+'               { MAIS }
| '='               { ATRIB }
| ':'               { DPONTOS }
| "<="              { MENORIGUAL }
| "->"              { SETA }
| "and"             { E   }
| int as num        { let valor = int_of_string num in LITINT valor }
| '"'               { let buf = Buffer.create 1 in
                      let s = leia_string buf lexbuf in LITSTRING s }
| "return"          { RETURN }
| "def"             { DEF }
| identificador as id { ID id }
| _ as c            { failwith (msg_erro lexbuf c); }
```

Mini analisador léxico para o Python : lexico.mll

```
and comentario_bloco n = parse
      "*/" { if n=0 then token lexbuf
             else comentario_bloco (n - 1) lexbuf }
    | "/*" { comentario_bloco (n + 1) lexbuf; }
    | _    { comentario_bloco n lexbuf }
    | eof  { failwith "Comentário não fechado" }

and leia_string buffer = parse
    | '"'      {Buffer.contents buffer }
    | "\\t"    { Buffer.add_char buffer '\t'; leia_string buffer
      lexbuf }
    | "\\n"    { Buffer.add_char buffer '\n'; leia_string buffer
      lexbuf }
    | '\\' '"' { Buffer.add_char buffer '"'; leia_string buffer
      lexbuf }
    | '\\' '\\' { Buffer.add_char buffer '\\'; leia_string buffer
      lexbuf }
    | _ as c   { Buffer.add_char buffer c; leia_string buffer lexbuf
       }
    | eof      { failwith "A string não foi fechada." }
```

O modulo carregador.ml

```
#load "lexico.cmo"
#load "pre_processador.cmo"

type nome_arq = string
type tokens = Lexico.token list

let rec tokens lexbuf =
  let tok = Pre_processador.lexico lexbuf in
  match tok with
  | Lexico.EOF -> ([Lexico.EOF]:tokens)
  | _ -> tok :: tokens lexbuf

let lexico str =
  let lexbuf = Lexing.from_string str in
  tokens lexbuf

let lex (arq:nome_arq) =
  let ic = open_in arq in
  let lexbuf = Lexing.from_channel ic in
  let toks = tokens lexbuf in
  let _ = close_in ic in toks
```

Compilação do analisador léxico para Python

- Para compilar, digite

```
ocamllex lexico.mll
ocamlc -c lexico.ml
ocamlc -c pre_processador.ml
```

- Para usar, entre no Ocaml

```
rlwrap ocaml
```

e digite

```
#use "carregador.ml";;
lex "teste.py";;
```