Universidade da Beira Interior

Desenho de Linguagens de Programação e de Compiladores

Simão Melo de Sousa

Aula 4 - Análise Léxica

Ihavetheabilitytoarrange1'sand0'sinsuchanorderthatanx86processorcanactually interpretandexecutethosecommands.It'scalledComputerProgramming,butit'sthe closestthatamancanevergettogivingbirthinmyopinion.AndIsomehowfeelresponsi bleforthefutureexistenceandacceptanceofmy"child".I'dspendhourstryingtofindth etinybugthatcausesmychildtomisbehaveoractstrangely.Butthat'smymildsuperpo wer...Imaketheworldabetterplacebywritingmindlessback-endprogramsthatno-on ewilleverseenorevenknowthatit'sthere.ButIknow;andthat'sallthatmatters.Stadium airconditioningfails-Fansprotest

I have the ability to arrange 1's and 0's in such an order that an x86 processor can actually interpret and execute those commands. It's called Computer Programming, but it's the closest that a man can ever get to giving birth in my opinion. And I somehow feel responsible for the future existence and acceptance of my "child". I'd spend hours trying to find the tiny bug that causes my child to misbehave or act strangely. But that's my mild superpower... I make the world a better place by writing mindless back-end programs that no-one will ever see nor even know that it's there. But I know; and that's all that matters.

atributed to Alucard

Stadium air conditioning fails — Fans protest

anonymous newspaper headline

a análise léxica define-se como o corte do texto fonte em « palavras »

à semelhança do que acontece em linguagem natural, este corte do texto em palavras facilita o trabalho da fase seguinte de análise do texto, a análise sintáctica

estas palavras, no contexto da análise léxica, são designadas de **lexemas** (*tokens*)

fonte = sequência de caracteres

```
fun x -> (* uma função *)
     x+1
```

↓

análise léxica

↓

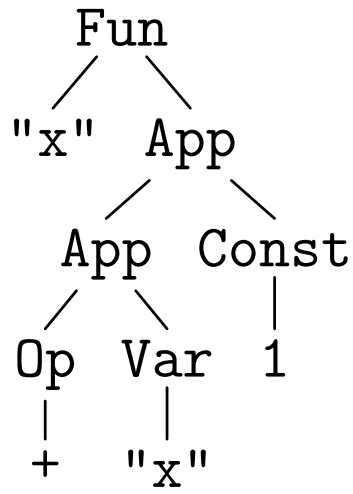
sequência de lexemas

⋮

análise sintáctica

(próxima aula)

sintaxe abstracta



os espaços/caracteres brancos (espaço, *carriage return*, tabulação, etc.) têm um papel de destaque na análise léxica ; permitam separar univocamente dois lexemas

assim `funx` é entendido como um só lexema (o identificador `funx`) e `fun x` é entendido como dois lexemas (a palavra chave `fun` e o identificador `x`)

em várias situações, no entanto, os caracteres brancos podem não ter utilidade (como em `x + 1`)
e podem simplesmente serem ignorados

os caracteres brancos não são contemplados no fluxo de lexemas enviados *em output* para as análises seguintes

as convenções diferem conforme as linguagens,
e determinados caracteres « brancos » podem ser significativos

exemplos :

- as tabulações para `make`
- os *carriage returns* e espaços em início de linha para Python ou Haskell (a indentação determina a estrutura dos blocos)

os comentários têm o papel de espaços em branco

```
fun(* aqui vai *)x -> x + (* junto um *) 1
```

aqui o comentário `(* aqui vai *)` toma o papel de um caractere branco significativo (separa dois lexemas) e o comentário `(* junto um *)` configura-se como um caractere branco não significativo

nota : os comentários são, por vezes, tratados por certas ferramentas (`ocamldoc`, `javadoc`, etc.), que os considera como informativos logo não são descartados por estas últimas, sendo processados nas respectivas análises léxicas

```
val length : 'a list -> int
  (** Return the length (number of elements) of ...
```

para realizar a análise léxica, vamos utilizar

- **expressões regulares** para descrever os lexemas
- **autómatos finitos** para reconhecê-los

vamos, em particular, tirar proveito da possibilidade de se construir automaticamente um autómato finito determinista minimal que reconhece a linguagem descrita por uma expressão regular (ver UC. de Teoria da Computação)

expressões regulares

assumimos a existência de um alfabeto A
o conjunto *RegExp* das expressões regulares r sobre A define-se por indução estrutural da forma seguinte

$$
\begin{array}{rcll}
r & ::= & \emptyset & \text{linguagem vazia} \\
 & | & \epsilon & \text{palavra vazia} \\
 & | & a & \text{caracter } a \in A \\
 & | & r\, r & \text{concatenação} \\
 & | & r \mid r & \text{alternativa} \\
 & | & r\star & \text{estrela - fecho de Kleene}
\end{array}
$$

convenção: a estrela tem a maior prioridade, seguida da concatenação e, finalmente, a alternativa

a **linguagem** definida pela expressão regular r é o conjunto das palavras $L(r)$ definida por recursão estrutural da seguinte forma

$$
\begin{aligned}
L(\emptyset) &= \emptyset \\
L(\epsilon) &= \{\epsilon\} \\
L(a) &= \{a\} \\
L(r_1\, r_2) &= \{w_1 w_2 \mid w_1 \in L(r_1) \wedge w_2 \in L(r_2)\} \\
L(r_1 \mid r_2) &= L(r_1) \cup L(r_2) \\
L(r\star) &= \textstyle\bigcup_{n\geq 0} L(r^n) \quad \text{onde } r^0 = \epsilon,\ r^{n+1} = r\, r^n
\end{aligned}
$$

considerando o alfabeto $\{a, b\}$

- palavras de três letras

$$(a|b)(a|b)(a|b)$$

- palavras que terminam com a letra a

$$(a|b) \star a$$

- palavras que alternam a e b

$$(b|\epsilon)(ab) \star (a|\epsilon)$$

constantes inteiras decimais, eventualmente precedidas de zeros

$$(0|1|2|3|4|5|6|7|8|9)\,(0|1|2|3|4|5|6|7|8|9)\star$$

identificadores compostos de uma letra, de algarismos e do *underscore*, começando necessariamente por uma letra

$$(a|b|\ldots|z|A|B|\ldots|Z)\,(a|b|\ldots|z|A|B|\ldots|Z|\_|0|1|\ldots|9)\star$$

constantes flutuantes de OCaml (`3.14` `2.` `1e-12` `6.02e23` etc.)

$$d\,d\star\,(.d\star\,|\,(\epsilon\,|\,.d\star)(e|E)\,(\epsilon|+|-)d\,d\star)$$

com $d = 0|1|\ldots|9$

os comentários da forma `(* ... *)`, mas **não aninhados**, podem igualmente serem definidos da seguinte forma

$$\boxed{(}\boxed{*}\ \left(\boxed{*}\star r_1 \mid r_2\right)\star\ \boxed{*}\boxed{*}\star\ \boxed{)}$$

onde r_1 = todos os caracteres excepto $*$ e)
e r_2 = todos os caracteres excepto $*$

as expressões regulares não são suficientemente expressivas para poder definir os comentários **aninhados** (a linguagem das palavras bem parenteseadas não é regular, é algébrico)

explicaremos mais adiante como contornar esta situação

dado o tipo

```
type expreg =
  | Empty
  | Epsilon
  | Char of char
  | Union   of expreg * expreg
  | Product of expreg * expreg
  | Star    of expreg
```

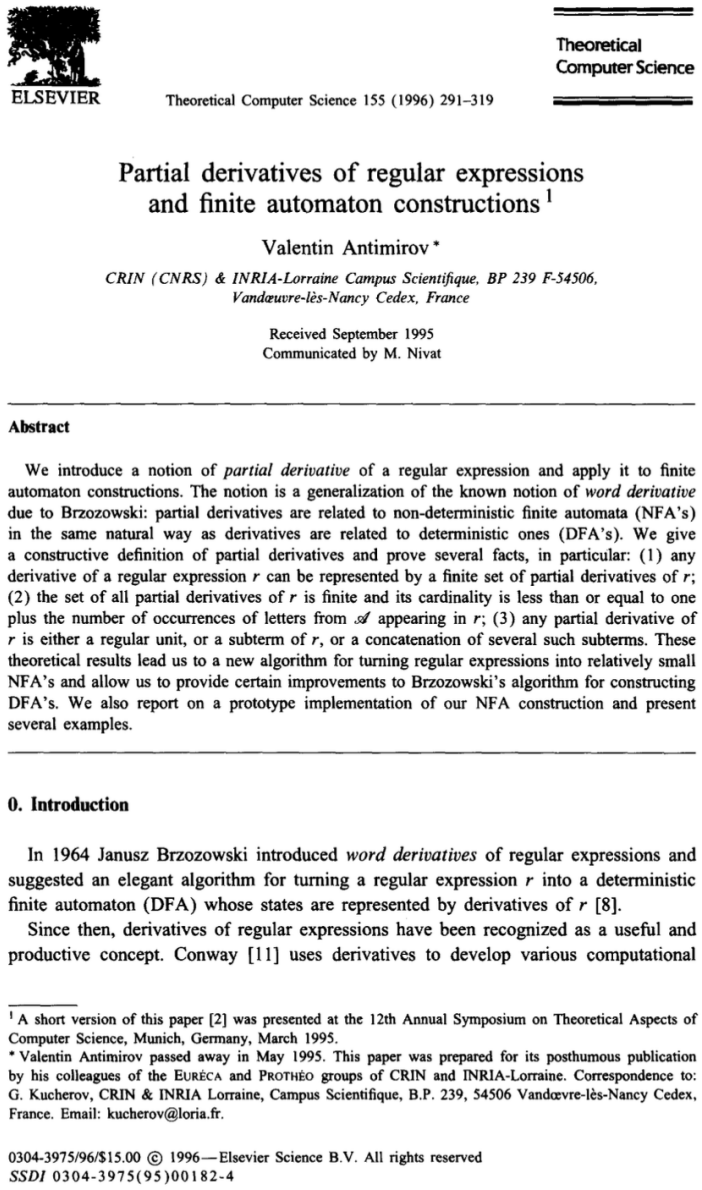
escrever uma função

```
val recognize: expreg -> char list -> bool
```

da forma mais simples possível

seja $\varepsilon(.) : \mathit{expreg} \rightarrow \mathbb{B}$ a função tal que:
$\varepsilon(r) = \mathit{true}$ se $\epsilon \in L(r)$,
$\varepsilon(r) = \mathit{false}$, senão.

dizemos que r é **anulável** quando
$\varepsilon(r) = \mathit{true}$



ELSEVIER

Theoretical Computer Science 155 (1996) 291–319

Theoretical Computer Science

Partial derivatives of regular expressions and finite automaton constructions [1]

Valentin Antimirov *

CRIN (CNRS) & INRIA-Lorraine Campus Scientifique, BP 239 F-54506, Vandœuvre-lès-Nancy Cedex, France

Received September 1995
Communicated by M. Nivat

Abstract

We introduce a notion of *partial derivative* of a regular expression and apply it to finite automaton constructions. The notion is a generalization of the known notion of *word derivative* due to Brzozowski: partial derivatives are related to non-deterministic finite automata (NFA's) in the same natural way as derivatives are related to deterministic ones (DFA's). We give a constructive definition of partial derivatives and prove several facts, in particular: (1) any derivative of a regular expression r can be represented by a finite set of partial derivatives of r; (2) the set of all partial derivatives of r is finite and its cardinality is less than or equal to one plus the number of occurrences of letters from $\mathscr{A}$ appearing in r; (3) any partial derivative of r is either a regular unit, or a subterm of r, or a concatenation of several such subterms. These theoretical results lead us to a new algorithm for turning regular expressions into relatively small NFA's and allow us to provide certain improvements to Brzozowski's algorithm for constructing DFA's. We also report on a prototype implementation of our NFA construction and present several examples.

0. Introduction

In 1964 Janusz Brzozowski introduced *word derivatives* of regular expressions and suggested an elegant algorithm for turning a regular expression r into a deterministic finite automaton (DFA) whose states are represented by derivatives of r [8].

Since then, derivatives of regular expressions have been recognized as a useful and productive concept. Conway [11] uses derivatives to develop various computational

[1] A short version of this paper [2] was presented at the 12th Annual Symposium on Theoretical Aspects of Computer Science, Munich, Germany, March 1995.

* Valentin Antimirov passed away in May 1995. This paper was prepared for its posthumous publication by his colleagues of the EURÉCA and PROTHÉO groups of CRIN and INRIA-Lorraine. Correspondence to: G. Kucherov, CRIN & INRIA Lorraine, Campus Scientifique, B.P. 239, 54506 Vandœvre-lès-Nancy Cedex, France. Email: kucherov@loria.fr.

SSDI 0304-3975(95)00182-4

para uma expressão regular r e un caractere c definimos

$$\partial_c(r) = \{w \mid cw \in L(r)\}$$

esta função calcula o que chamamos de **derivada (parcial) de r em relação a c** (também designado de **resíduo** de r em c)

$$\partial_c(r) = \begin{cases} \emptyset & se\ r = \emptyset \vee r = \epsilon \vee (r = b \wedge b \neq c) \\ \epsilon & se\ r = c \\ \partial_c(u) + \partial_c(v) & se\ r = u + v \\ \partial_c(u)v + \partial_c(v) & se\ r = uv \wedge \epsilon(u) = true \\ \partial_c(u)v & se\ r = uv \wedge \epsilon(u) = false \\ \partial_c(u)u\star & se\ r = u\star \end{cases}$$

esta função estende-se naturalmente para palavras
$\partial_\epsilon(r) = r$ e $\partial_{wa}(r) = \partial_a(\partial_w(r))$

Teorema: Dada uma palavra w, e uma expressão regular r,

$$\epsilon(\partial_w(r)) = true \Leftrightarrow w \in L(r)$$

como definir a função $\varepsilon(r)$?

```
let rec null = function
  | Empty | Char _ -> false
  | Epsilon | Star _ -> true
  | Union   (r1, r2) -> null r1 || null r2
  | Product (r1, r2) -> null r1 && null r2
```

e a função $\partial_c(r)$?

```
let rec derive r c  = match r with
  | Empty | Epsilon  -> Empty
  | Char d           -> if c = d then Epsilon else Empty
  | Union (r1, r2)   -> Union (derive r1 c, derive r2 c)
  | Product (r1, r2) -> let r' = Product (derive r1 c, r2) in
                        if null r1 then Union (r', derive r2 c)
                                   else r'
  | Star r1          -> Product (derive r1 c, r)
```

```
let rec recognize r = function
  | []      -> null r
  | c :: w -> recognize (derive r c) w
```

(20 linhas!)

um exemplo

```
let algarismos =
 Union (Char '0',
 Union (Char '1',
 Union (Char '2',
 Union (Char '3',
 Union (Char '4',
 Union (Char '5',
 Union (Char '6',
 Union (Char '7',
 Union (Char '8', Char '9'))))))))

let inteiros =
    Product (algarismos, Star algarismos)
```

```
# recognize inteiros [];;
- : bool = false
# recognize inteiros ['0'];;
- : bool = true
# recognize inteiros ['0';'0';'9';'0'];;
- : bool = true
# (* com um 'o' maiúscula *)
  recognize inteiros ['0';'0';'9';'O'];;
- : bool = false
# recognize inteiros ['0';'0'];;
- : bool = true
```

o módulo `Str` da biblioteca standard oferece utilitários baseados no processamento de expressões regulares

em particular, oferece um tipo para as expressões regulares : `regexp`

```
type regexp
```

oferece uma função que transforma uma string (que obedece a um certo formato) na expressão regular subjacente

```
(* val regexp : string -> regexp *)
# let inteiros = regexp "[0-9]+";;
val inteiros : Str.regexp = <abstr>
```

oferece funções de reconhecimento

```
(* val string_match : regexp -> string -> int -> bool
   val matched_string : string -> string *)
 let match_regexp r s =
       let _ =  string_match r s 0 in (matched_string s) = s
```

```
# match_regexp inteiros "00089898900" ;;
- : bool = true
# match_regexp inteiros "000898989OO" ;;
- : bool = false
```

e muito mais outros processamentos úteis com base nestas expressões regulares...

```
# Str.split (regexp "[\t\n ]+") "It's 5.50 a.m....
 Do you know where your\t\t stack pointer is?";;
- : string list = ["It's"; "5.50"; "a.m...."; "Do"; "you";
"know"; "where"; "your"; "stack"; "pointer"; "is?"]
```

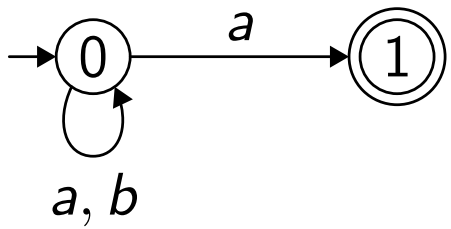
autómatos finitos

Definição

Um autómato finito R sobre um alfabeto A é um quadruplo (Q, T, I, F) onde

- *Q é um conjunto finito de estados*
- *$T \subseteq Q \times A \times Q$ um conjunto de transições*
- *$I \subseteq Q$ um conjunto de estados **iniciais***
- *$F \subseteq Q$ um conjunto de estados **terminais***

exemplo : $Q = \{0, 1\}$, $T = \{(0, a, 0), (0, b, 0), (0, a, 1)\}$, $I = \{0\}$, $F = \{1\}$



uma palavra $a_1 a_2 \ldots a_n \in A^\star$ é **reconhecido** por um autómato (Q, T, I, F) se e só se

$$s_0 \xrightarrow{a_1} s_1 \xrightarrow{a_2} s_2 \cdots s_{n-1} \xrightarrow{a_n} s_n$$

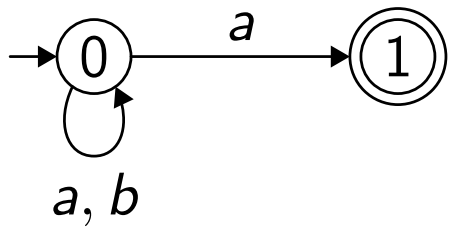
com $s_0 \in I$, $(s_{i-1}, a_i, s_i) \in T$ para todos i, e $s_n \in F$

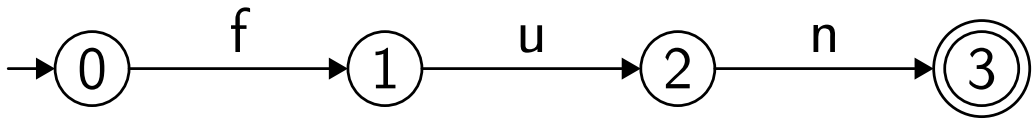
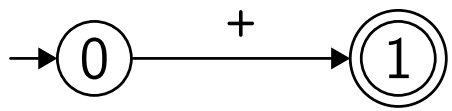
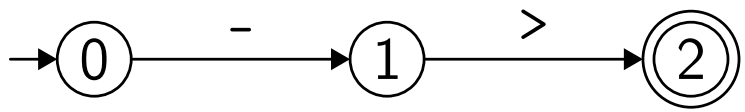
a **linguagem** $L(R)$ definida por um autómato R é o conjunto das palavras reconhecidas

Teorema (de Kleene)

As expressões regulares e os autómatos finitos definem as mesmas linguagens: ***as linguagens regulares***

$(a|b) \star a$

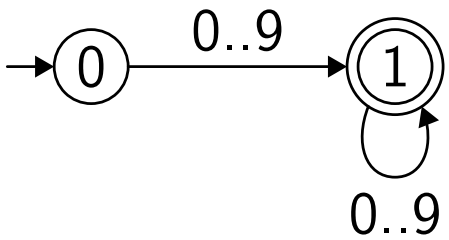


lexema	expressão régular	autómato
palavra chave `fun`	*f u n*	
símbolo +	+	
símbolo ->	- >	

expressão regular

$$(0|1|2|3|4|5|6|7|8|9)\,(0|1|2|3|4|5|6|7|8|9)\star$$

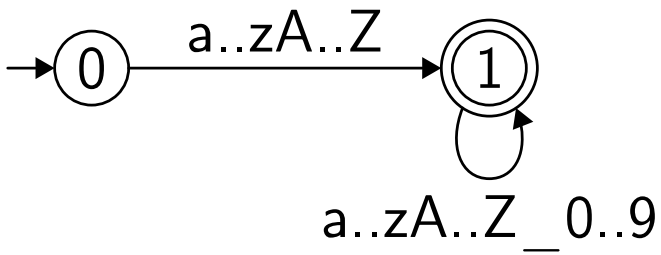
autómato



expressão regular

$$(a|b|\ldots|z|A|B|\ldots|Z)\,(a|b|\ldots|z|A|B|\ldots|Z|\_|0|1|\ldots|9)\star$$

autómato

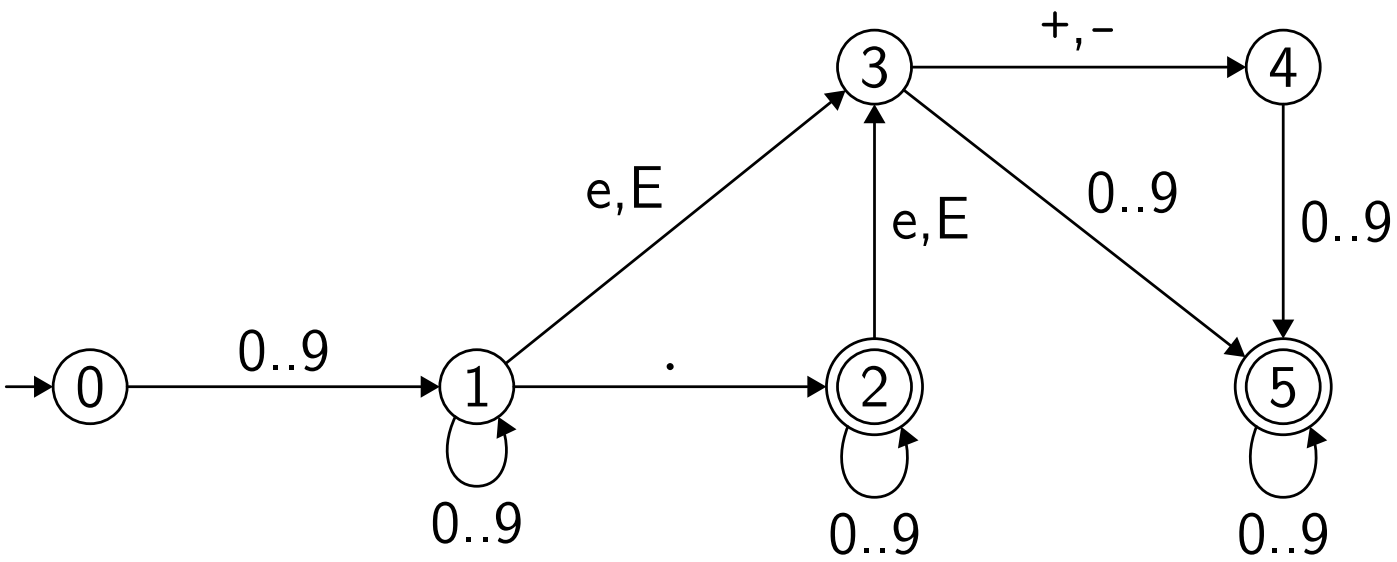


expressão regular

$$d\,d\star\,(.d\star\,|\,(\epsilon\,|\,.d\star)(e|E)\,(\epsilon|+|-)d\,d\star)$$

onde $d = 0|1|\ldots|9$

autómato

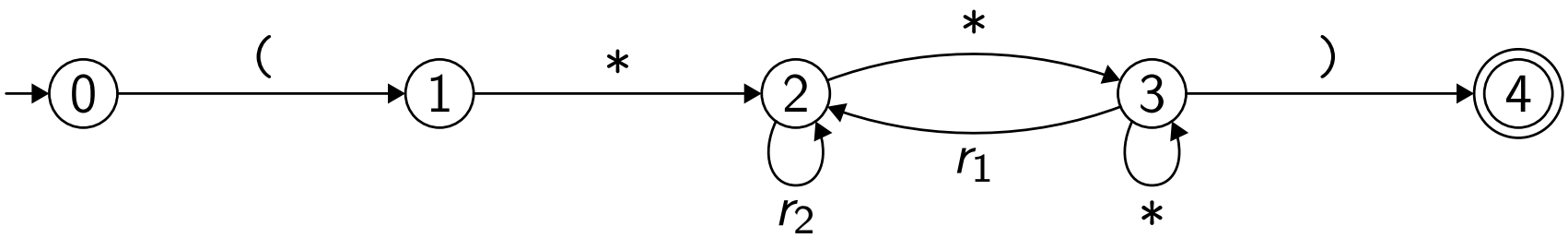


expressões regular

$$\boxed{(}\ \boxed{*}\ \left(\boxed{*} \star r_1 \mid r_2\right) \star\ \boxed{*}\ \boxed{*} \star\ \boxed{)}$$

onde r_1 = todos os caracteres excepto $*$ e)
e r_2 = todos os caracteres excepto $*$

autómato finito



analisador léxico

um **analisador léxico** é um autómato finito para a « **união** » de todas as expressões regulares definindo os lexemas

o funcionamento do analisador léxico, *no entanto*, difere do simples reconhecimento da palavra por um autómato, porque

- é necessário decompor a palavra em entrada (*a fonte*) numa **sequência** de palavras reconhecidas
- pode haver **ambiguidades**
- é necessário construir lexemas (os estados finais executam **acções**)

a palavra `funx` é reconhecido pela expressão regular dos identificadores, mas contém um prefixo reconhecido por uma outra expressão regular (`fun`)

$\Rightarrow$ *escolhemos* reconhecer o lexema que representa/consome **a maior quantidade de caracteres** possível da fonte

a palavra `fun` é reconhecida pela expressão regular associada a palavra chave `fun` mas também pela expressão regular dos identificadores

$\Rightarrow$ *classificamos* os lexemas por **ordem de prioridade**

com as três expressões regulares

$$a, \quad ab, \quad bc$$

um analisador léxico irá **falhar** com a entrada

$$abc$$

(*ab* é reconhecida, como a mais comprida, e em seguida falha sobre *c*)

apesar da palavra *abc* pertencer à linguagem $(a|ab|bc)^+$

o analisador léxico deve então memorizar o último estado final atingido, se necessário

quando não há mais transições possíveis, uma de duas situações :

- nenhuma posição final foi memorizada $\Rightarrow$ falha da análise léxica
- leu-se o prefixo wv na entrada, sendo w o lexema reconhecido pelo último estado final pelo qual se passou $\Rightarrow$ retornamos w e reinicia-se a análise com v prefixado ao resto da entrada

programememos um analizador léxico...

comecemos por introduzir um tipo para representar os autómatos finitos deterministas

```
type automaton = {
  initial : int;                  (* estado = inteiro *)
  trans   : int Cmap.t array;     (* estado -> char -> estado *)
  action  : action array;         (* estado -> acção *)
}
```

com

```
type action =
  | NoAction          (* sem acção = estado não final *)
  | Action of string  (* natureza do lexema *)
```

e

```
module Cmap = Map.Make(Char)
```

a tabela de transição é cheia para os estados (vector) e é esparsa para os caracteres (AVL)

damo-nos o código seguinte

```
let transition autom state c =
  try Cmap.find c autom.trans.(state) with Not_found -> -1
```

o objectivo é escrever uma função `analyzer` que toma um autómato e uma cadeia de caracteres por analisar e retorna uma função que calcula o próximo lexema

ou seja

```
val analyzer :
  automaton -> string -> (unit -> string * string)
```

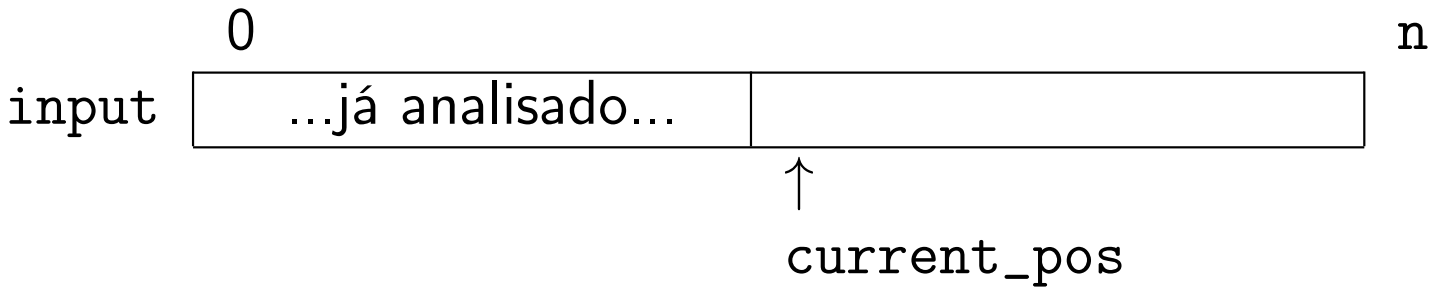
nota : poderíamos igualmente devolver a lista dos lexemas, mas *adotamos* aqui a metodologia em uso na interação entre analise léxica e analise sintáctica

utilização :

```
# let next_token = analyzer autom "fun funx";;
# next_token ();;

- : string * string = ("keyword", "fun")

# next_token ();;

- : string * string = ("space", " ")

# next_token ();;

- : string * string = ("ident", "funx")

# next_token ();;

Exception: Failure "Lexical Error".
```

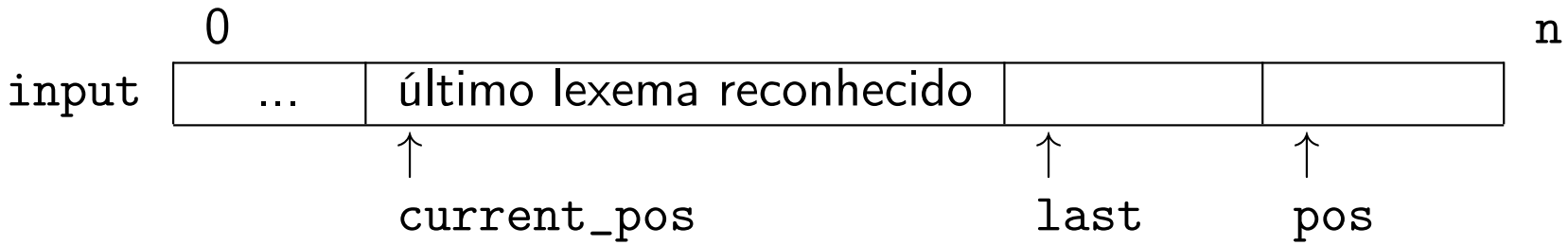
memorizamos a posição atual na entrada com a ajuda de uma referência `current_pos`

```
let analyzer autom input =
  let n = String.length input in
  let current_pos = ref 0 in (* posição actual *)
  fun () ->
    ...
```



nota : a aplicação parcial de `analyzer` é fulcral ! (... ex: veja a função `next_token`)

```
let analyzer autom input =
  let n = String.length input in (* n e current_pos *)
  let current_pos = ref 0 in     (* no fecho de scan *)
  fun () ->
    let rec scan last state pos =
      (* pronto para analizar o caractere pos *)
      ...
    in
    scan None autom.initial !current_pos
```



determinamos então se uma transição é possível

```
let rec scan last state pos =
  let state' =
    if pos = n then -1
    else transition autom state input.[pos]
  in
  if state' >= 0 then
    (* uma transição para state' *) ...
  else
    (* sem transição possível *) ...
```

no caso positivo, actualizamos `last`, se for necessário, e chamamos `scan` recursivamente sobre `state'`

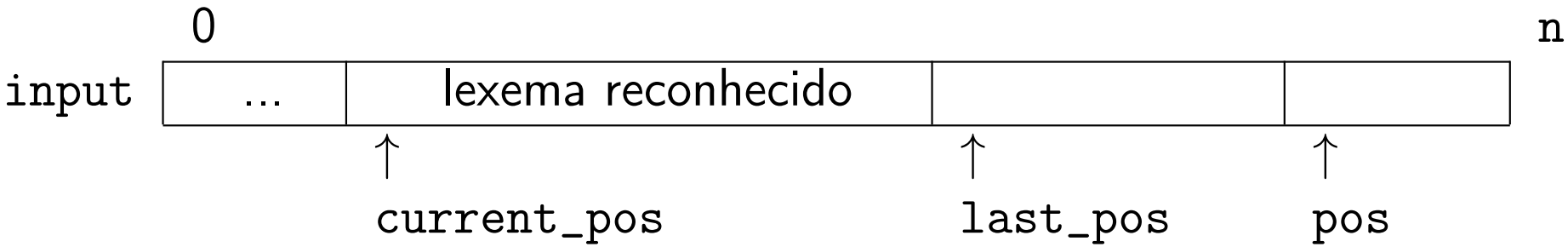
```
    if state' >= 0 then
      let last = match autom.action.(state') with
        | NoAction -> last
        | Action a -> Some (pos + 1, a)
      in
      scan last state' (pos + 1)
```

`last` tem a forma `Some (p, a)` onde

- `p` é a posição que segue o lexema reconhecido
- `a` é a acção (o tipo do lexema)

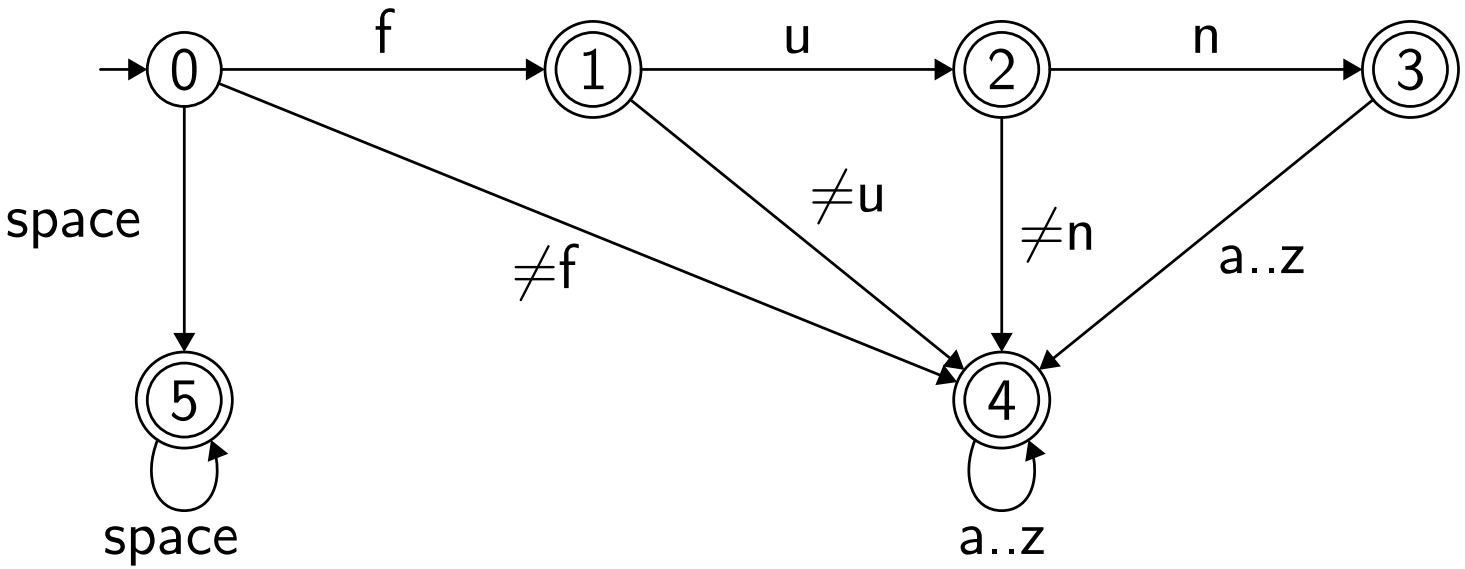
se, pelo contrário, nenhuma transição é possível, examinamos o `last` para determinar o resultado

```
    else match last with
     | None ->
         failwith "Lexical Error"
     | Some (last_pos, action) ->
         let start = !current_pos in
         current_pos := last_pos;
         action, String.sub input start (last_pos - start)
```



testemos o código com

- uma palavra chave : *fun*
- identificadores : $(a..z)(a..z)\star$
- caracteres brancos : *space space*$\star$



```
let autom = {
  initial = 0;
  trans = [| ... |];
  action = [|
    (*0*) NoAction;
    (*1*) Action "ident";
    (*2*) Action "ident";
    (*3*) Action "keyword";
    (*4*) Action "ident";
    (*5*) Action "space";
  |]
}
```

```
# let next_token = analyzer autom "fun funx";;
# next_token ();;

- : string * string = ("keyword", "fun")

# next_token ();;

- : string * string = ("space", " ")

# next_token ();;

- : string * string = ("ident", "funx")

# next_token ();;

Exception: Failure "Lexical Error".
```

há, claro, várias outras formas de programar um analisador léxico

exemplo : *n* funções mutuamente recursivas, uma por cada estado do autómato

```
let rec state0 pos = match input.[pos] with
  | 'f'                  -> state1 (pos + 1)
  | 'a'..'e' | 'g'..'z' -> state4 (pos + 1)
  | ' '                  -> state5 (pos + 1)
  | _                    -> failwith "Lexical Error"

and state1 pos = match input.[pos] with
  | ...
```

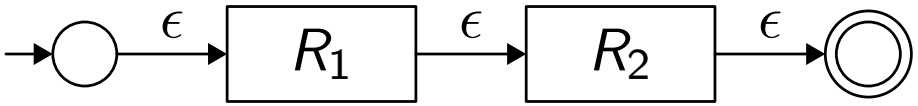
(exploraremos alguns detalhes desta abordagem nas aulas práticas)

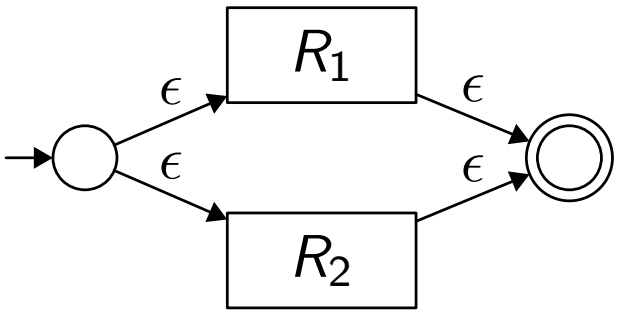
em prática, dispomos de ferramentas que constroem analizadores léxicos a partir das suas descrições com base em expressões regulares e respectivas acções

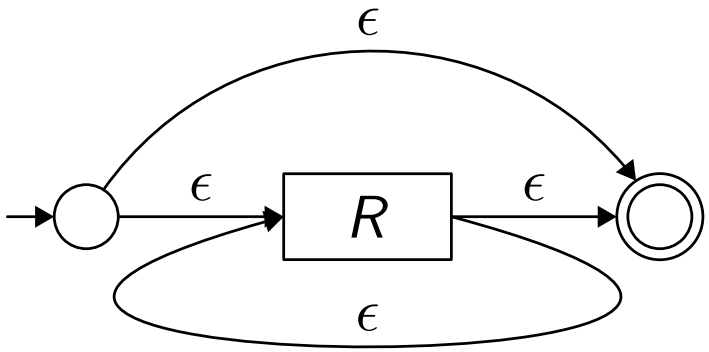
é a grande família de `lex` : `lex`, `flex`, `jflex`, `ocamllex`, etc.

Construção do autómato

podemos construir um autómato finito correspondente a uma expressão regular passando pelo intermédio de um autómato não determinista







podemos depois determinisar, e até em seguida, minimisar o autómato resultante

foi este mesmo processo que foi explorado em *Teoria da Computação*

mas...

. . . podemos também construir **directamente** um autómato determinista

o ponto de partida : podemos pôr em correspondência as letras de uma palavra reconhecida com as que aparecem na expressão regular

exemplo : a palavra *aabaab* é reconhecida por $(a|b) \star a(a|b)$, da forma seguinte

$$
\begin{array}{ll}
a & (\mathbf{a}|b) \star a(a|b) \\
a & (\mathbf{a}|b) \star a(a|b) \\
b & (a|\mathbf{b}) \star a(a|b) \\
a & (\mathbf{a}|b) \star a(a|b) \\
a & (a|b) \star \mathbf{a}(a|b) \\
b & (a|b) \star a(a|\mathbf{b})
\end{array}
$$

distingamos as diferentes letras da expressão regular :

$$(a_1|b_1) \star a_2(a_3|b_2)$$

vamos construir um autómato cujos estados são conjuntos de letras

o estado s reconhece as palavras cujas primeiras letras pertencem a s

exemplo : o estado $\{a_1, a_2, b_1\}$ reconhece as palavras cuja primeira letra é ou um a correspondente a a_1 ou a a_2, ou um b correspondente a b_1

para construir as transições de s_1 para s_2 é necessário estabelecer as letras que podem aparecer a seguir a uma determina letra, numa palavra reconhecida (*follow*)

exemplo : sempre com $r = (a_1|b_1) \star a_2(a_3|b_2)$, temos

$$follow(a_1, r) = \{a_1, a_2, b_1\}$$

para calcular *follow*, precisamos do cálculo das primeiras (resp. últimas) letras possíveis de uma palavra reconhecida (*first*, resp. *last*)

exemplo : sempre com $r = (a_1|b_1) \star a_2(a_3|b_2)$, temos

$$\begin{aligned} \mathit{first}(r) &= \{a_1, a_2, b_1\} \\ \mathit{last}(r) &= \{a_3, b_2\} \end{aligned}$$

para calcular *first* e *last*, precisamos de um último conceito : será que a palavra vazia pertence à linguagem reconhecida ? (*null*)

$$
\begin{aligned}
null(\emptyset) &= \texttt{false} \\
null(\epsilon) &= \texttt{true} \\
null(a) &= \texttt{false} \\
null(r_1\, r_2) &= null(r_1) \wedge null(r_2) \\
null(r_1 \mid r_2) &= null(r_1) \vee null(r_2) \\
null(r\star) &= \texttt{true}
\end{aligned}
$$

(reconhecemos aqui um conceito explorado mais acima...)

podemos então definir *first*

$$
\begin{aligned}
first(\emptyset) &= \emptyset \\
first(\epsilon) &= \emptyset \\
first(a) &= \{a\} \\
first(r_1\, r_2) &= first(r_1) \cup first(r_2) \quad \text{se } null(r_1) \\
&= first(r_1) \quad \text{senão} \\
first(r_1 \mid r_2) &= first(r_1) \cup first(r_2) \\
first(r\star) &= first(r)
\end{aligned}
$$

a definição de *last* é similar (**deixado em execício!**)

seguimos com *follow*

$$
\begin{aligned}
follow(c, \emptyset) &= \emptyset \\
follow(c, \epsilon) &= \emptyset \\
follow(c, a) &= \emptyset \\
follow(c, r_1\, r_2) &= follow(c, r_1) \cup follow(c, r_2) \cup first(r_2) \quad \text{se } c \in last(r_1) \\
&= follow(c, r_1) \cup follow(c, r_2) \quad \text{senão} \\
follow(c, r_1 \mid r_2) &= follow(c, r_1) \cup follow(c, r_2) \\
follow(c, r\star) &= follow(c, r) \cup first(r) \quad \text{se } c \in last(r) \\
&= follow(c, r) \quad \text{senão}
\end{aligned}
$$

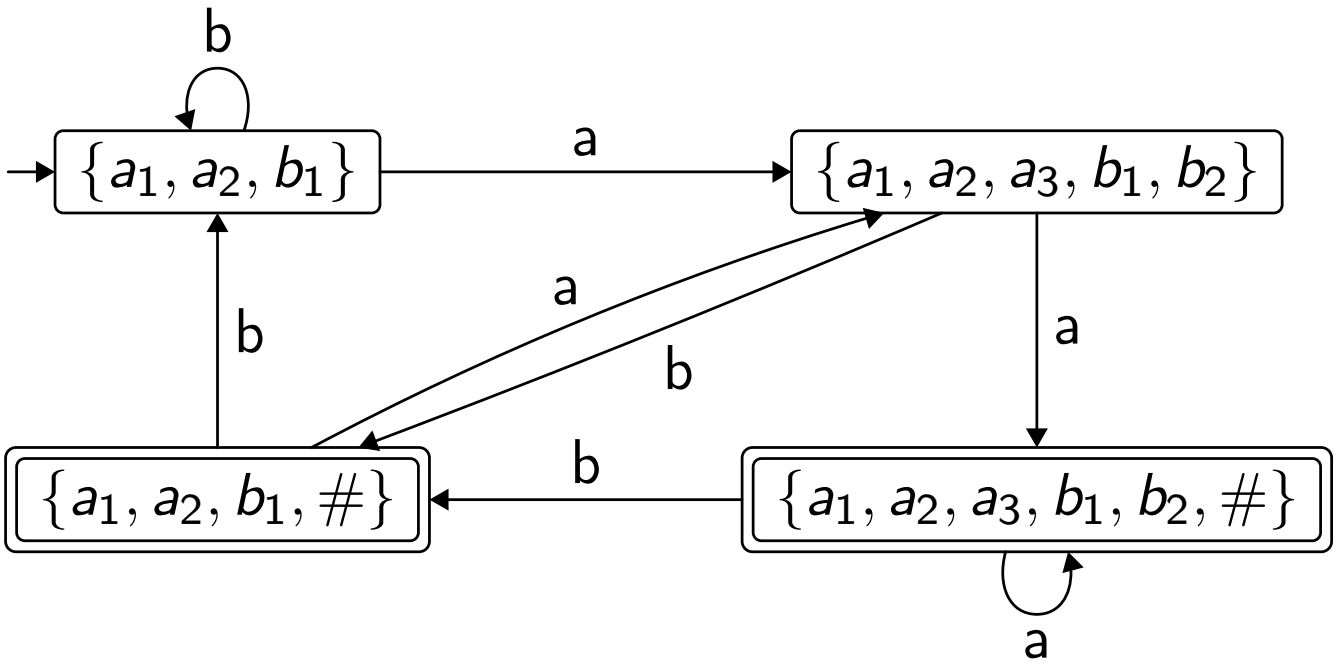
enfim, podemos construir o autómato para a expressão regular r

começamos por pós-fixar r com o caractere # para assinalar o seu fim

o algoritmo é então o seguinte:

1. o estado inicial é o conjunto $first(r\#)$
2. enquanto existir um estado s para o qual é necessário calcular as transições
   para cada caractere c do alfabeto
   seja s' o estado $\bigcup_{c_i \in s} follow(c_i, r\#)$
   juntar a transição $s \xrightarrow{c} s'$
3. os estados finais são os estados contendo #

para $(a|b) \star a(a|b)$ obtemos



(mais sobre este algoritmo, e a sua implementação, nas aulas práticas)

para $r \triangleq (a|b|c) \star b(a|b)\star$

- reescrever a expressão regular com as suas componentes indexadas
- calcular as funções `null`, `first`, `last` e `follow`
- usando o algoritmo de construção directa, determinar o autómato que reconhece r

a biblioteca Genlex

Genlex é um módulo `OCaml` da biblioteca standard para criar rapidamente analisadores léxicos para linguagens simples

a documentação está disponível aqui (link)

pressupõe que os lexemas possam todos ser classificados em 6 categorias:

- palavras chave (`Kwd`)
- identificadores (`Ident`)
- números inteiros (`Int`)
- números flutuantes (`Float`)
- cadeias de caracteres (`String`)
- caracteres (`Char`)

Module Genlex

```
module Genlex: sig .. end
```

A generic lexical analyzer.

This module implements a simple 'standard' lexical analyzer, presented as a function from character streams to token streams. It implements roughly the lexical conventions of OCaml, but is parameterized by the set of keywords of your language.

Example: a lexer suitable for a desk calculator is obtained by

```
let lexer = make_lexer ["+";"-";"*";"/";"let";"="; "("; ")"]
```

The associated parser would be a function from `token stream` to, for instance, `int`, and would have rules such as:

```
let rec parse_expr = parser
  | [< n1 = parse_atom; n2 = parse_remainder n1 >] -> n2
and parse_atom = parser
  | [< 'Int n >] -> n
  | [< 'Kwd "("; n = parse_expr; 'Kwd ")" >] -> n
and parse_remainder n1 = parser
  | [< 'Kwd "+"; n2 = parse_expr >] -> n1+n2
  | [< >] -> n1
```

One should notice that the use of the `parser` keyword and associated notation for streams are only available through camlp4 extensions. This means that one has to preprocess its sources *e. g.* by using the `"-pp"` command-line switch of the compilers.

```
type token =
| Kwd of string
| Ident of string
| Int of int
| Float of float
| String of string
| Char of char
```

Um exemplo simples

```
open Genlex
open Stream

let keywords =
(* serve para destacar as palavras chaves dos identificadores*)
  [ "LET"; "PRINT";  "IF"; "THEN"; "ELSE";  "+";  "*"; "/";  "=" ]

let line_lexer l = Genlex.make_lexer keywords (Stream.of_string l)
```

Um exemplo simples

```
# let tokenstream = line_lexer "LET x = x + y * 3" ;;
val tokenstream : Genlex.token Stream.t = <abstr>
# Stream.next tokenstream;;
- : Genlex.token = Kwd "LET"
# next tokenstream;;
- : Genlex.token = Ident "x"
# next tokenstream;;
- : Genlex.token = Kwd "="
# next tokenstream;;
- : Genlex.token = Ident "x"
# next tokenstream;;
- : Genlex.token = Kwd "+"
# next tokenstream;;
- : Genlex.token = Ident "y"
# next tokenstream;;
- : Genlex.token = Kwd "*"
# next tokenstream;;
- : Genlex.token = Int 3
# next tokenstream;;
Exception: Stream.Failure.
```

a ferramenta ocamllex

um ficheiro `ocamllex` tem por sufixo `.mll` e tem a forma seguinte

```
{
  ... código OCaml arbitrário ...
}
rule f1 = parse
| regexp1 { acção1 }
| regexp2 { acção2 }
| ...
and f2 = parse
  ...
and fn = parse
  ...
{
  ... código OCaml arbitrário ...
}
```

compila-se o ficheiro `lexer.mll` com `ocamllex`

```
% ocamllex lexer.mll
```

tem por efeito a produção de um ficheiro OCaml `lexer.ml` que contém a definição de uma função para cada analizador `f1`, ..., `fn` :

```
val f1 : Lexing.lexbuf -> type1
val f2 : Lexing.lexbuf -> type2
...
val fn : Lexing.lexbuf -> typen
```

o tipo Lexing.lexbuf

o tipo `Lexing.lexbuf` é o tipo da estrutura de dados que contém o estado dum analizador léxico

o módulo `Lexing` da biblioteca standard fornece vários meios para construir um valor deste tipo

```
val from_channel : Pervasives.in_channel -> lexbuf

val from_string : string -> lexbuf

val from_function : (string -> int -> int) -> lexbuf
```

_ (underscore)	qualquer caractere
`'a'`	o caractere `'a'`
`"foobar"`	a string `"foobar"` (em particular ϵ =)
[*caracteres*]	conjunto de caracteres (por ex. `['a'-'z' 'A'-'Z']`)
[^*caracteres*]	o complemento (por ex. `[^ '"']` - tudo menos...)
r_1 | r_2	a alternativa
$r_1\ r_2$	a concatenação
r *	a estrela/fecho de Kleene
r +	uma ou várias repetições de r ($\stackrel{\text{def}}{=} r\ r\star$)
r ?	uma ou zero ocorrências de r ($\stackrel{\text{def}}{=} \epsilon \mid r$)
`eof`	fim da entrada

identificadores

```
  | ['a'-'z' 'A'-'Z'] ['a'-'z' 'A'-'Z' '_' '0'-'9']* { ... }
```

constantes inteiras

```
  | ['0'-'9']+ { ...}
```

constantes flutuantes

```
  | ['0'-'9']+
      ( '.' ['0'-'9']*
      | ('.' ['0'-'9']*)? ['e' 'E'] ['+' '-']?  ['0'-'9']+ )
    { ... }
```

podemos definir atalhos para expressões regulares

```
let letter = ['a'-'z' 'A'-'Z']
let digit = ['0'-'9']
let decimals = '.' digit*
let exponent = ['e' 'E'] ['+' '-']?  digit+

rule token = parse
  | letter (letter | digit | '_')*          { ... }

  | digit+                                  { ... }

  | digit+ (decimals | decimals? exponent)  { ... }
```

para analizadores definidos pela palavra chaves `parse`, aplica-se a regra da escolha do lexema que consome o maior texto da entrada

em caso, mesmo assim, de empate, é dada prioridade à regra que aparece primeiro na definição do analizador

```
| "fun"          { print_endline "função"}
| ['a'-'z']+     { print_endline ("identificador:  ") }
```

para escolher a regra contrária (o menor consumo da entrada), basta utilizar a palavra chave `shortest` no lugar de `parse`

```
rule scan = shortest
  | regexp1 { acção1 }
  | regexp2 { acção2 }
  ...
```

mecanismo original (presente desde as primeiras versões)

```
| ['a'-'z']+    {print_endline (Lexing.lexeme lexbuf) }
```

em alternativa,

podemos dar um nome à string reconhecida, ou a sub-strings que correspondem a sub-expressões regulares, com a ajuda da construção as

```
| ['a'-'z']+ as s                              { print_endline s }

| (['+' '-']? as sign) (['0'-'9']+ as n) {print_endline (sign^n)}
```

numa acção, é possível invocar recursivamente o analizador léxico, ou um qualquer outro analizador léxico ali definido

o buffer da analise léxica deve ser passado em argumento ;
está contido - como já o vimos em exemplos anteriores - numa variável com o nome `lexbuf`

é assim muito fácil e cómodo tratar dos caracteres brancos :

```
rule token = parse
  | [' ' '\t' '\n']+ { token lexbuf }
  | ...
```

para tratar dos comentários, podemos utilizar uma expressão regular

... ou um analizador léxico dedicado :

```
rule token = parse
  | "(*"  { comment lexbuf }
  | ...

and comment = parse
  | "*)"  { token lexbuf }
  | _     { comment lexbuf }
  | eof   { failwith "comentário inacabado" }
```

vantagem : tratamos de forma correcta o erro relacionado com um comentário inacabado

outra vantagem : tratamos de forma simples os **comentários aninhados**

com um contador

```
rule token = parse
  | "(*"  { level := 1; comment lexbuf; token lexbuf }
  | ...

and comment = parse
  | "*)"  { decr level; if !level > 0 then comment lexbuf }
  | "(*"  { incr level; comment lexbuf }
  | _     { comment lexbuf }
  | eof   { failwith "comentário inacabado" }
```

... ou até mesmo sem contador !

```
rule token = parse
  | "(*"  { comment lexbuf; token lexbuf }
  | ...

and comment = parse
  | "*)"  { () }
  | "(*"  { comment lexbuf; comment lexbuf }
  | _     { comment lexbuf }
  | eof   { failwith "comentário inacabado" }
```

nota : percebemos com este exemplo que ultrapassamos o poder das expressões regulares com esta construção

dado o tipo OCaml para os lexemas

```
type token =
  | Tident of string
  | Tconst of int
  | Tfun
  | Tarrow
  | Tplus
  | Teof
```

```
rule token = parse
  | [' ' '\t' '\n']+ { token lexbuf }
  | "(*"             { comment lexbuf }
  | "fun"            { Tfun }
  | ['a'-'z']+ as s  { Tident s }
  | ['0'-'9']+ as s  { Tconst (int_of_string s) }
  | "+"              { Tplus }
  | ->"              { Tarrow }
  | _ as c           { failwith ("caractere inválido : " ^
                                 String.make 1 c) }
  | eof              { Teof }

and comment = parse
  | "*)"  { token lexbuf }
  | _     { comment lexbuf }
  | eof   { failwith "comentário inacabado" }
```

ocamllex propõe um conjunto mínimo de mecanismos para auxiliar o programador na gestão dos locais onde ocorrem eventuais erros

a estrutura lexbuf dispõe do campo

```
type lexbuf = { ... mutable lex_curr_p : position;...}
```

de tipo position

```
type position = {
      (* nome do ficheiro analizado *)
      pos_fname : string;
      (*contador das linhas - número da linha corrente*)
      pos_lnum : int;
     (* posição absoluta do inicio da linha corrente *)
      pos_bol : int;
     (* posição absoluta da posição corrente, em #caracteres *)
      pos_cnum : int;  }
```

por omissão, ocamllex só trata da atualização do campo pos_cnum

é assim necessário inicializar os restantes campos e actualiza-los devidamente quando necessário

```
(* função que faz reset às posições no lx (de tipo lexbuf) *)
(* com base no nome do ficheiro f       *)
let init_pos lx f =  let pos = lx.lex_curr_p in
    lx.lex_curr_p <-
      {pos with pos_fname = f ; pos_lnum = 0 ; pos_bol = pos.pos_cnum}
```

para a actualização podemo-nos auxiliar da função `new_line` (do módulo `Lexing`)

```
rule token = parse
       ...
 | [' ' '\t']+ { token lexbuf }
 | '\n'        { Lexing.newline lexbuf ; token lexbuf }
       ...
 | _ as c      { let pos  = lexbuf.lex_curr_p in
                 let line = string_of_int (pos.pos_lnum) in
                 let col  = string_of_int (pos.pos_lnum - pos.pos_bol) in
                   failwith ("caractere inválido : " ^(String.make 1c)^
                     " linha : "^line^" coluna : "^col)}
       ...
```

quatro « regras » que não podemos esquecer quando escrevemos um analizador léxico

1. tratar dos **caracteres brancos**
2. as regras **prioritárias em primeiro** (por ex. palavras chaves antes dos identificadores)
3. assinalar os **erros léxicos** (caracteres inválidos, mas também comentários ou strings mal fechados, sequências de escape inválidas, etc.)
4. tratar do **fim de entrada** (`eof`)

por omissão `ocamllex` codifica o autómato numa **tabela**, que é interpretada à execução

a opção `-ml` permite a produção de código `OCaml` puro, ondeo autómato é implementado por funções ; esta solução nao é recomendada na prática

mesmo utilizando uma tabela, o autómato pode ter uma tamanho demasiado grande, em particular se a linguagem contempla um número grande de palavras chaves

é preferível usar uma só expressão regular para capturar os identificadores **e** as palavras chaves, e separá-las dentro da acção correspondente com recurso a uma tabela de palavras-chaves

```
{
  let keywords = Hashtbl.create 97
  let () = List.iter (fun (s,t) -> Hashtbl.add keywords s t)
    ["and", AND; "as", AS; "assert", ASSERT;
     "begin", BEGIN; ...
}
rule token = parse
| ident as s
  { try Hashtbl.find keywords s with Not_found -> IDENT s }
```

se desejamos que um analizador não seja *case-sensitive* então
evitar absolutamente

```
| ("a"|"A") ("n"|"N") ("d"|"D")
    { AND }
| ("a"|"A") ("s"|"S")
    { AS }
| ("a"|"A") ("s"|"S") ("s"|"S") ("e"|"E") ("r"|"R") ("t"|"T")
    { ASSERT }
| ...
```

preferir

```
rule token = parse
| ident as s
  { let s = String.lowercase s in
    try Hashtbl.find keywords s with Not_found -> IDENT s }
```

para compilar (ou recompilar) os módulos Ocaml, é preciso estabelecer as **dependências** entre estes módulos, com recurso a `ocamldep`

contudo, `ocamldep` não conhece a sintaxe `ocamllex` ⇒ devemos assegurar da geração prévia do código `ocaml` com `ocamllex`

o `Makefile` pode ser algo como :

```
lexer.ml: lexer.mll
	ocamllex lexer.mll

.depend: lexer.ml
	ocamldep *.ml *.mli > .depend

include .depend
```

alternativa : usar `ocamlbuild`

aplicações de ocamllex

(para além da análise léxica)

a utilização de ocamllex não está limitada a análise léxica

quando se pretende analisar um texto (string, ficheiro, fluxo, sequência de pacotes formatados de informação, etc.) com base em expressões regulares, ocamllex é uma ferramenta particularmente adaptada

em particular para escrever **filtros**, *i.e.* programas que traduzem uma linguagem para uma outra via modificações locais e relativamente simples

exemplo 1 : agrupar várias linhas em branco consecutivas numa só

tão simples como

```
rule scan = parse
  | '\n' '\n'+ { print_string "\n\n"; scan lexbuf }
  | _ as c     { print_char c; scan lexbuf }
  | eof        { () }

{ let () = scan (Lexing.from_channel stdin) }
```

o processo de compilação pode ser o seguinte

```
% ocamllex mbl.mll
% ocamlopt -o mbl mbl.ml
```

a utilização segue o padrão seguinte

```
% ./mbl < infile > outfile
```

exemplo 2 : contar as ocorrências de uma determinada palavra num texto

```
{
  let word = Sys.argv.(1)
  let count = ref 0
}
rule scan = parse
  | ['a'-'z' 'A'-'Z']+ as w
     { if word = w then incr count; scan lexbuf }
  | _
     { scan lexbuf }
  | eof
     { () }
{
  let () = scan (Lexing.from_channel (open_in Sys.argv.(2)))
  let () = Printf.printf "%d occurrence(s)\n" !count
}
```

exemplo 3 : um pequeno tradutor de Ocaml para HTML,
para embelezar código fonte numa visualização web/online

objetivo

- uso : `caml2html file.ml`, que produz `file.ml.html`
- palavras chaves a verde, comentários a vermelho
- numerar as linhas
- isto tudo em menos de 100 linhas de código

escrevemos tudo num só ficheiro `caml2html.mll`

começamos por verificar as opções da linha de comando

```
{
  let () =
    if Array.length Sys.argv <> 2
    || not (Sys.file_exists Sys.argv.(1)) then begin
      Printf.eprintf "usage: caml2html file\n";
      exit 1
    end
```

em seguida abrimos o ficheiro HTML em modo escrita e o actualizamos com recurso à função `fprintf`

```
  let file = Sys.argv.(1)
  let cout = open_out (file ^ ".html")
  let print s = Printf.fprintf cout s
```

escrevemos o início do ficheiro HTML definindo como título o nome do ficheiro
utilizamos a tag HTML <pre> para formatar o código de acordo com a formatação original

```
let () =
  print «html><head><title>%s</title></head><body>\n<pre>" file
```

introduzimos uma função para numerar cada linha,
e invocamo-la imediatamente para a primeira linha

```
let count = ref 0
let newline () = incr count; print "\n%3d: " !count
let () = newline ()
```

definimos uma tabela de palavras-chaves (como no caso da análise léxica)

```
  let is_keyword =
    let ht = Hashtbl.create 97 in
    List.iter
      (fun s -> Hashtbl.add ht s ())
      [ "and"; "as"; "assert"; "asr"; "begin"; "class";
        ... ];
    fun s -> Hashtbl.mem ht s
}
```

introduzimos uma expressão regular para os identificadores

```
let ident =
  ['A'-'Z' 'a'-'z' '_'] ['A'-'Z' 'a'-'z' '0'-'9' '_']*
```

podemos agora tratar do analizador em si

para um identificador, testamos se se trata de uma palavra-chave

```
rule scan = parse
  | ident as s
      { if is_keyword s then begin
          print «font color=\"green\»%s</font>" s
        end else
          print "%s" s;
        scan lexbuf }
```

a cada fim de linha, invocamos a função `newline` :

```
  | "\n"
      { newline (); scan lexbuf }
```

para um comentário, mudamos a cor (vermelho) e invocamos outro analizador `comment` especializado na análise de comentários ; aquando do seu retorno, retomamos a cor por omissão e a análise por `scan`

```
  | "(*"
      { print «font color=\"990000\»(*";
        comment lexbuf;
        print «/font>";
        scan lexbuf }
```

qualquer outro caractere é impresso tal e qual

```
  | _ as s { print "%s" s; scan lexbuf }
```

presenciando o fim da entrada, terminamos

```
  | eof    { () }
```

para a análise dos comentários, basta não esquecer o processamento de `newline` :

```
and comment = parse
  | "(*"    { print "(*"; comment lexbuf; comment lexbuf }
  | "*)"    { print "*)" }
  | eof     { () }
  | "\n"    { newline (); comment lexbuf }
  | _ as c { print "%c" c; comment lexbuf }
```

termina-se com a aplicação do analizador `scan` sobre o ficheiro de entrada

```
{
  let () =
    scan (Lexing.from_channel (open_in file));
    print «/pre>\n</body></html>\n";
    close_out cout
}
```

está *quase* certo... :

- o caractere < tem significado em HTML, a sua tradução de OCaml deve assim ser `<`
- da mesma forma é preciso traduzir & para `&`

- uma string OCaml pode conter `(*`
  (como é precisamente o caso neste programa !)

corrijamos !

juntamos uma regra para < e um analizador para as strings

```
| ‘‘<"     { print "&lt;"; scan lexbuf }
| "&"      { print "&amp;"; scan lexbuf }
| ’"’      { print "\; string lexbuf; scan lexbuf }
```

é preciso ter em atenção ao caractere ’"’, quando " não indica o início de uma string

```
| "’\"’"
| _ as s { print "%s" s; scan lexbuf }
```

reflectimos este tratamento ao processo dos comentários (de facto, em OCaml podemos comentar código contendo "*)")

```
  | '"'     { print "\; string lexbuf; comment lexbuf }
  | "'\"'"
  | _ as s { print "%s" s; comment lexbuf }
```

finalmente, as strings são tratadas pelo analizador `string`, sem nos esquecermos das sequências de escape (tais como \" por exemplo)

```
and string = parse
  | '"'     { print "\ }
  | «"     { print "&lt;"; string lexbuf }
  | "&"     { print "&amp;"; string lexbuf }
  | "\\" _
  | _ as s { print "%s" s; string lexbuf }
```

agora sim, funciona como pretendido
(um bom teste é experimentar sobre o próprio `caml2html.mll`)

para ser perfeito, deveríamos também converter as tabulações em início de linha (tipicamente inseridos pelos editores de texto) como espaços

a alteração é deixada como exercício...

exemplo 4 : indentação automática de programas C

ideia :

- em cada abertura de chaveta, aumentamos a tabulação
- em cada fecho de chaveta, diminuímo-la
- em cada fim de linha, imprimimos a tabulação actual
- sem esquecer o processamento das strings e dos comentários

dados os meios de manter o estado das tabulações e das respectivas visualizações

```
{
  open Printf

  let margin = ref 0
  let print_margin () =
    printf "\n"; for i = 1 to 2 * !margin do printf " " done
}
```

em cada fim de linha, ignoramos os espaços em início de linha e mostramos a tabulação actual

```
let space = [' ' '\t']

rule scan = parse
  | '\n' space*
      { print_margin (); scan lexbuf }
```

as chavetas modificam a tabulação corrente

```
  | "{"
      { incr margin; printf "{"; scan lexbuf }
  | "}"
      { decr margin; printf "}"; scan lexbuf }
```

casos particulares : um fecho de chaveta em início de linha deve diminuir a tabulação antes da sua produção

```
  | '\n' space* "}"
      { decr margin; print_margin (); printf "}";
        scan lexbuf }
```

não esqueçamos as strings

```
  | '"' ([^ '\\' '"'] | '\\' _)* '"' as s
      { printf "%s" s; scan lexbuf }
```

nem os comentários numa linha

```
  | "//" [^ '\n']* as s
      { printf "%s" s; scan lexbuf }
```

nem os comentários da forma /* ... */

```
  | "/*"
      { printf "/*"; comment lexbuf; scan lexbuf }
```

onde

```
and comment = parse
  | "*/"
      { printf "*/" }
  | eof
      { () }
  | _ as c
      { printf "%c" c; comment lexbuf }
```

para terminar, qualquer outra situação é reproduzida tal e qual

`eof` termina a análise

```
rule scan = parse
  | ...
  | _ as c
      { printf "%c" c; scan lexbuf }
  | eof
      { () }
```

o programa principal cabe em duas linhas

```
{
  let c = open_in Sys.argv.(1)
  let () = scan (Lexing.from_channel c); close_in c
}
```

nesta forma, ainda há obviamente margem para melhorias

em particular, um corpo de um ciclo ou de uma condicional está reduzida a uma só instrução não sofrerá indentação suplementar :

```
if (x==1)
     continue;
```

exercício : replicar este exercício para OCaml (é bastante mais difícil, mas poderá limitar-se a alguns casos simples)

conclusão

- as **expressões regulares** estão na base da análise léxica
- o trabalho de analise pode ser em grande parte automatizado com a utilização de ferramentais tais como `ocamllex`
- a capacidade expressiva de `ocamllex` ultrapassa a das expressões regulares, e pode assim ser utilizado muito para além da análise léxica

  (nota : estes mesmos acetatos foram escritos com a ajuda de um pre-processador de LaTeX escrito com a ajuda de `ocamllex`)

- aulas práticas
  - compilação de expressão regulares para os autómatos finitos
- próxima aula teórica
  - análise sintáctica (primeira parte)

estes acetatos resultam essencialmente de uma adaptação do material pedagógico gentilmente cedido pelo Jean-Christophe Filliâtre (link1, link2)

adicionalmente poderá consultar as obras seguintes

- **Modern Compilers: Principles, Techniques, and Tools**, Alfred V. Aho, Monica S. Lam, Ravi Sethi et Jeffrey D. Ullman
- **Types and Programming Languages**, Benjamin C. Pierce
- **Modern Compiler Implementation**, Andrew W. Appel (3 versões: ML, C, Java)