UNIVERSIDADE
BEIRA INTERIOR

Teoria da Computação
Aula - OCamllex e Menhir

Simão Melo de Sousa

introdução

uma exploração prática de ferramentas que se sustentam sobre as linguagens regulares e algébricas, os seus conceitos, e os seus algoritmos

- a biblioteca genlex
- a biblioteca str
- a ferramenta ocamllex
- a ferramenta ocamlyacc
- a ferramenta menhir

análise léxica, uma curta introdução

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

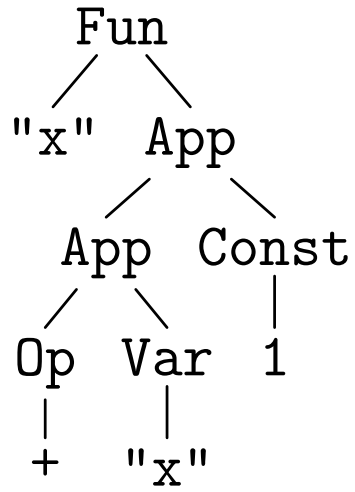
```
fun x -> (* uma função *)
     x+1
```

análise léxica

sequência de lexemas

análise sintáctica

sintaxe abstracta



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 biblioteca Str

Str é um módulo `OCaml` da biblioteca standard para criar rapidamente processadores de texto baseados em expressões regulares

a documentação está disponível aqui (link)

Chapter 29 The str library: regular expressions and string processing

The `str` library provides high-level string processing functions, some based on regular expressions. It is intended to support the kind of file processing that is usually performed with scripting languages such as `awk`, `perl` or `sed`.

Programs that use the `str` library must be linked as follows:

```
ocamlc other options str.cma other files
ocamlopt other options str.cmxa other files
```

For interactive use of the `str` library, do:

```
ocamlmktop -o mytop str.cma
./mytop
```

or (if dynamic linking of C libraries is supported on your platform), start `ocaml` and type `#load "str.cma";;`.

- Module `Str`: regular expressions and string processing

Str é um módulo `OCaml` da biblioteca standard para criar rapidamente processadores de texto baseados em expressões regulares

a documentação está disponível aqui (link)

Module Str

```
module Str: sig .. end
```
Regular expressions and high-level string processing

Regular expressions

```
type regexp
```
The type of compiled regular expressions.

```
val regexp : string -> regexp
```
Compile a regular expression. The following constructs are recognized:

- `.` Matches any character except newline.
- `*` (postfix) Matches the preceding expression zero, one or several times
- `+` (postfix) Matches the preceding expression one or several times
- `?` (postfix) Matches the preceding expression once or not at all
- `[..]` Character set. Ranges are denoted with `-`, as in `[a-z]`. An initial `^`, as in `[^0-9]`, complements the set. To include a `]` character in a set, make it the first character of the set. To include a `-` character in a set, make it the first or the last character of the set.
- `^` Matches at beginning of line: either at the beginning of the matched string, or just after a '\n' character.
- `$` Matches at end of line: either at the end of the matched string, or just before a '\n' character.
- `\|` (infix) Alternative between two expressions.
- `\(..\)` Grouping and naming of the enclosed expression.
- `\1` The text matched by the first `\(...\)` expression (`\2` for the second expression, and so on up to `\9`).
- `\b` Matches word boundaries.
- `\` Quotes special characters. The special characters are `$^\.*+?[]`.

Note: the argument to `regexp` is usually a string literal. In this case, any backslash character in the regular expression must be doubled to make it past the OCaml string parser. For example, the following expression:

```
let r = Str.regexp "hello \\([A-Za-z]+\\)" in
    Str.replace_first r "\\1" "hello world"
```

returns the string `"world"`.

In particular, if you want a regular expression that matches a single backslash character, you need to quote it in the argument to `regexp` (according to the last item of the list above) by adding a second backslash. Then you need to quote both backslashes (according to the syntax of string constants in OCaml) by doubling them again, so you need to write four backslash characters: `Str.regexp "\\\\"`.

```
val regexp_case_fold : string -> regexp
```
Same as `regexp`, but the compiled expression will match text in a case-insensitive way: uppercase and lowercase letters will be considered equivalent.

```
val quote : string -> string
```
`Str.quote s` returns a regexp string that matches exactly `s` and nothing else.

Str é um módulo `OCaml` da biblioteca standard para criar rapidamente processadores de texto baseados em expressões regulares

a documentação está disponível aqui (link)

String matching and searching

`val string_match : regexp -> string -> int -> bool`

`string_match r s start` tests whether a substring of `s` that starts at position `start` matches the regular expression `r`. The first character of a string has position `0`, as usual.

`val search_forward : regexp -> string -> int -> int`

`search_forward r s start` searches the string `s` for a substring matching the regular expression `r`. The search starts at position `start` and proceeds towards the end of the string. Return the position of the first character of the matched substring.
Raises `Not_found` if no substring matches.

`val search_backward : regexp -> string -> int -> int`

`search_backward r s last` searches the string `s` for a substring matching the regular expression `r`. The search first considers substrings that start at position `last` and proceeds towards the beginning of string. Return the position of the first character of the matched substring.
Raises `Not_found` if no substring matches.

`val string_partial_match : regexp -> string -> int -> bool`

Similar to `Str.string_match`, but also returns true if the argument string is a prefix of a string that matches. This includes the case of a true complete match.

`val matched_string : string -> string`

`matched_string s` returns the substring of `s` that was matched by the last call to one of the following matching or searching functions:

- `Str.string_match`
- `Str.search_forward`
- `Str.search_backward`
- `Str.string_partial_match`
- `Str.global_substitute`
- `Str.substitute_first`

provided that none of the following functions was called in between:

- `Str.global_replace`
- `Str.replace_first`
- `Str.split`
- `Str.bounded_split`
- `Str.split_delim`
- `Str.bounded_split_delim`
- `Str.full_split`
- `Str.bounded_full_split`

Note: in the case of `global_substitute` and `substitute_first`, a call to `matched_string` is only valid within the `subst` argument, not after `global_substitute` or `substitute_first` returns.

The user must make sure that the parameter `s` is the same string that was passed to the matching or searching function.

Replacement

`val global_replace : regexp -> string -> string -> string`

`global_replace regexp templ s` returns a string identical to `s`, except that all substrings of `s` that match `regexp` have been replaced by `templ`. The replacement template `templ` can contain `\1`, `\2`, etc; these sequences will be replaced by the text matched by the corresponding group in the regular expression. `\0` stands for the text matched by the whole regular expression.

`val replace_first : regexp -> string -> string -> string`

Same as `Str.global_replace`, except that only the first substring matching the regular expression is replaced.

`val global_substitute : regexp -> (string -> string) -> string -> string`

`global_substitute regexp subst s` returns a string identical to `s`, except that all substrings of `s` that match `regexp` have been replaced by the result of function `subst`. The function `subst` is called once for each matching substring, and receives `s` (the whole text) as argument.

`val substitute_first : regexp -> (string -> string) -> string -> string`

Same as `Str.global_substitute`, except that only the first substring matching the regular expression is replaced.

`val replace_matched : string -> string -> string`

`replace_matched repl s` returns the replacement text `repl` in which `\1`, `\2`, etc. have been replaced by the text matched by the corresponding groups in the regular expression that was matched by the last call to a matching or searching function (see `Str.matched_string` for details). `s` must be the same string that was passed to the matching or searching function.

Splitting

`val split : regexp -> string -> string list`

`split r s` splits `s` into substrings, taking as delimiters the substrings that match `r`, and returns the list of substrings. For instance, `split (regexp "[ \t]+") s` splits `s` into blank-separated words. An occurrence of the delimiter at the beginning or at the end of the string is ignored.

`val bounded_split : regexp -> string -> int -> string list`

Same as `Str.split`, but splits into at most `n` substrings, where `n` is the extra integer parameter.

`val split_delim : regexp -> string -> string list`

Same as `Str.split` but occurrences of the delimiter at the beginning and at the end of the string are recognized and returned as empty strings in the result. For instance, `split_delim (regexp " ") " abc "` returns `[""; "abc"; ""]`, while `split` with the same arguments returns `["abc"]`.

`val bounded_split_delim : regexp -> string -> int -> string list`

Same as `Str.bounded_split`, but occurrences of the delimiter at the beginning and at the end of the string are recognized and returned as empty strings in the result.

```
type split_result =
| Text of string
| Delim of string
```

`val full_split : regexp -> string -> split_result list`

Same as `Str.split_delim`, but returns the delimiters as well as the substrings contained between delimiters. The former are tagged `Delim` in the result list; the latter are tagged `Text`. For instance, `full_split (regexp "[{}]") "{ab}"` returns `[Delim "{"; Text "ab"; Delim "}"]`.

`val bounded_full_split : regexp -> string -> int -> split_result list`

Same as `Str.bounded_split_delim`, but returns the delimiters as well as the substrings contained between delimiters. The former are tagged `Delim` in the result list; the latter are tagged `Text`.

```
(* um pequeno utilitário *)
let (=~) s re = Str.string_match (Str.regexp re) s 0
```

```
# let pad = "[ACGT]*TGC";;
val pad : string = "[ACGT]*TGC"
# let st = "ATTGCAGTAGGACTCGCCTGATGCAGTC";;
- : string = "ATTGCAGTAGGACTCGCCTGATGCAGTC"
# st =~ pad;;
- : bool = true
# matched_string st;;
- : string = "ATTGCAGTAGGACTCGCCTGATGC"
```

outro exemplo

```
let portuguese_date_of d =
  match d with
   [mm; dd; aa] -> dd^"/"^mm^"/"^aa
  | _ -> failwith "Bad date format"

(* atenção: string cortada para os efeitos da aula *)
let english_date_format = Str.regexp
"\\([0-9][0-9]\\)\\.\\([0-9][0-9]\\)\\.\\(\\([0-9][0-9]\\)
\\|\\([0-9][0-9][0-9][0-9]\\)\\)"

let trans_date l =
  try
   let  _ =Str.search_forward english_date_format l 0 in
   let d1 = Str.matched_string l in
   let d2 = portuguese_date_of (Str.split (Str.regexp "\.") d1) in
     Str.global_replace english_date_format d2 l
  with Not_found -> l

let _ = print_string (trans_date ".....08.19..99.....01.27.09...........\n")
```

```
# let _ = print_string (trans_date ".....08.19..99.....01.27.09...........") ;;
.....08.19..99.....27/01/09...........
- : unit = ()
```

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 $\epsilon = $)
[*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" }
```

localização dos erros

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)

não é só o léxico que importa, a sintaxe e a morfologia da frase também

Man eating piranha mistakenly sold as pet fish

Babies are what the mother eats

- objectivos da análise sintáctica é reconhecer as frases pertencendo à sintaxe da linguagem

- que o fluxo de caracteres é para a análise léxica, o fluxo dos lexemas – resultado desta mesma análise – o são para a análise sintáctica

- a saída da análise sintáctica é uma árvore de sintaxe abstracta

sequência de caracteres

`"fun x -> ( x + 1 )"`

↓

análise léxica

↓

sequência de lexemas

`fun`	`x`	`->`	`(`	`x`	`+`	`1`	`)`

sequência de lexemas

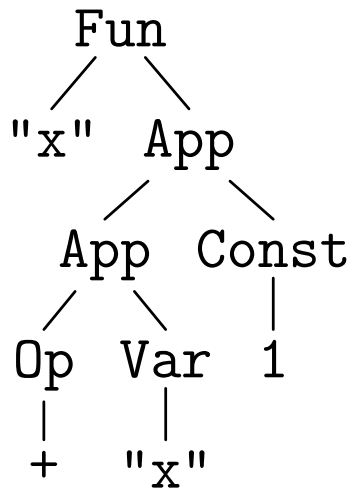
`fun`	`x`	`->`	`(`	`x`	`+`	`1`	`)`

↓

análise sintáctica

↓

sintaxe abstracta



OCamlyacc

nesta aula vamos dar mais ênfase à ferramenta menhir

a introdução ao ocamlyacc far-se-á em consequência por uma explicação prática dos problemas encontrados na **engenharia das gramáticas** LALR(1) (que se estendem naturalmente às gramáticas LR(1), mais permissivas no que diz respeito ao risco de conflito)

todos os exemplos apresentados são compatíveis com o menhir (podemos usar o menhir no lugar do ocamlyacc sobre os mesmos ficheiros apresentados)

```
%{
(*  Cabeçalho.  Opcional, pode ser omitido. *)
(*  Contém eventual código OCaml *)
%}
%token INT PLUS MINUS TIMES DIV EOF
%start expr
%type <unit> expr
%%

expr: INT              { (* acção - código OCaml *) }
    | expr PLUS expr   {}
    | expr MINUS expr  {}
    | expr TIMES expr  {}
    | expr DIV expr    {}
    | MINUS expr       {};

%%
  (* Trailer. Opcional : pode ser omitido.*)
  (* Contém eventual código OCaml *)
```

precedência e associatividade - (ocamlyacc -v calc.mly)

```
   0  $accept : %entry% $end
   1  expr : INT
   2       | expr PLUS expr
   3       | expr MINUS expr
   4       | expr TIMES expr
   5       | expr DIV expr
   6       | MINUS expr
   7  %entry% : '\001' expr
(.....)
14: shift/reduce conflict (shift 7, reduce 5) on PLUS
14: shift/reduce conflict (shift 8, reduce 5) on MINUS
14: shift/reduce conflict (shift 9, reduce 5) on TIMES
14: shift/reduce conflict (shift 10, reduce 5) on DIV
state 14
expr : expr . PLUS expr  (2)
expr : expr . MINUS expr  (3)
expr : expr . TIMES expr  (4)
expr : expr . DIV expr  (5)
expr : expr DIV expr .  (5)

PLUS  shift 7
MINUS  shift 8
TIMES  shift 9
DIV  shift 10
$end  reduce 5
(.....)
State 14 contains 4 shift/reduce conflicts.
9 terminals, 3 nonterminals
8 grammar rules, 15 states
```

precedência e associatividade: a solução

```
%{
%}
%token INT PLUS MINUS TIMES DIV EOF
%left PLUS MINUS
%left TIMES DIV
%left uminus
%start expr
%type <unit> expr
%%

expr: INT                                {}
    | expr PLUS expr                     {}
    | expr MINUS expr                    {}
    | expr TIMES expr                    {}
    | expr DIV expr                      {}
    | MINUS expr %prec uminus            {};
```

dangling else

Consideremos as três regras: (1) $S \rightarrow \mathit{if}\ E\ \mathit{then}\ S\ \mathit{else}\ S$,
(2) $S \rightarrow \mathit{if}\ E\ \mathit{then}\ S$ e (3) $S \rightarrow \mathit{resto}$

- Em $\mathit{if}\ a\ \mathit{then}\ \mathit{if}\ b\ \mathit{then}\ x\ \mathit{else}\ y \implies$ a que *if* pertence o *else*?
- Conflito *shift*/*reduce* com *else* entre
  (*reduce*) $S \rightarrow \mathit{if}\ E\ \mathit{then}\ S\ .$ e (*shift*) $S \rightarrow \mathit{if}\ E\ \mathit{then}\ S\ .\ \mathit{else}\ S$
- uma solução: reescrever a gramática desta forma
  1 $S \rightarrow M$
  2 $S \rightarrow U$
  3 $M \rightarrow \mathit{if}\ E\ \mathit{then}\ M\ \mathit{else}\ M$
  4 $M \rightarrow \mathit{resto}$
  5 $U \rightarrow \mathit{if}\ E\ \mathit{then}\ S$
  6 $U \rightarrow \mathit{if}\ E\ \mathit{then}\ M\ \mathit{else}\ U$
- outra solução: deixar como está! De facto em caso de conflito *shift*/*reduce*, as ferramentas como yacc assumem a interpretação *shift*.
- Este truque deve ser usado com cuidado! Em caso de conflito verificar com cuidado se podem deixar o yacc decidir sempre pelo *shift*.

em concreto...(ocamlyacc -v exemplo.mly)

```
%{
%}

%token ID WHILE BEGIN END DO IF THEN ELSE SEMI ASSIGN EOF
%start prog
%type <unit>prog
%%

prog: stmlist EOF              {};

stmlist: stm                   {}
       | stmlist SEMI stm      {};

stm: ID ASSIGN ID              {}
   | WHILE ID DO stm           {}
   | BEGIN stmlist END         {}
   | IF ID THEN stm            {}
   | IF ID THEN stm ELSE stm   {};
```

em concreto...(ocamlyacc -v exemplo.mly)

```
   0  $accept : %entry% $end
   1  prog : stmlist EOF
   2  stmlist : stm
   3          | stmlist SEMI stm
   4  stm : ID ASSIGN ID
   5      | WHILE ID DO stm
   6      | BEGIN stmlist END
   7      | IF ID THEN stm
   8      | IF ID THEN stm ELSE stm
   9  %entry% : '\001' prog
(......)
22: shift/reduce conflict (shift 23, reduce 7) on ELSE
state 22
stm : IF ID THEN stm .  (7)
stm : IF ID THEN stm . ELSE stm  (8)

ELSE  shift 23
END  reduce 7
SEMI  reduce 7
EOF  reduce 7
(......)
State 22 contains 1 shift/reduce conflict.
14 terminals, 5 nonterminals
10 grammar rules, 25 states
```

conflitos *reduce*/*reduce*

```
%token ID PLUS MINUS AND OR EQUAL ASSIGN EOF
%left OR
%left AND
%left PLUS
%start stm
%type <unit> stm
%%

stm: ID ASSIGN ae     {}
   | ID ASSIGN be     {};

be: be OR be          {}
  | be AND be         {}
  | ae EQUAL ae       {}
  | ID                {};

ae: ae PLUS ae        {}
  | ID                {};
```

conflitos *reduce*/*reduce*

```
   0  $accept : %entry% $end
   1  stm : ID ASSIGN ae
   2      | ID ASSIGN be
   3  be : be OR be
   4     | be AND be
   5     | ae EQUAL ae
   6     | ID
   7  ae : ae PLUS ae
   8     | ID
(.....)
6: reduce/reduce conflict (reduce 6, reduce 8) on $end
state 6
be : ID .  (6)
ae : ID .  (8)

$end  reduce 6
PLUS  reduce 8
AND  reduce 6
OR  reduce 6
EQUAL  reduce 8
(.....)
State 6 contains 1 reduce/reduce conflict.
11 terminals, 5 nonterminals
10 grammar rules, 19 states
```

aqui a dificuldade é que um identificador pode ocorrer tanto numa expressão booleana (*be*) como numa expressão aritmética (*ae*)

na analise dum identificador, que produção escolher para a redução? a regra 6 (identificador booleano) ou a regra 8 (identificador aritmético)?

nesta fase da compilação não temos forma fácil de inferir em que situação nos encontramos. Por isso o mais sensato é reescrever a gramática de tal forma que a escolha seja feita posteriormente: na **análise semântica** (tipagem, análise de porte etc.)

conflitos *reduce*/*reduce*

```
%token ID PLUS MINUS AND OR EQUAL ASSIGN EOF
%left OR
%left AND
%left PLUS
%start stm
%type <unit> stm
%%

stm: ID ASSIGN expr     {};

expr: expr OR expr      {}
    | expr AND expr     {}
    | expr EQUAL expr   {}
    | expr PLUS expr    {}
    | ID                {};
```

(Documentação do OCamlYacc)

The precedence declarations are used in the following way to resolve reduce/reduce and shift/reduce conflicts:

- Tokens and rules have precedences. By default, the precedence of a rule is the precedence of its rightmost terminal. You can override this default by using the %*prec* directive in the rule.
- A reduce/reduce conflict is resolved in favor of the first rule (in the order given by the source file), and ocamlyacc outputs a warning.
- A shift/reduce conflict is resolved by comparing the precedence of the rule to be reduced with the precedence of the token to be shifted. If the precedence of the rule is higher, then the rule will be reduced; if the precedence of the token is higher, then the token will be shifted.
- A shift/reduce conflict between a rule and a token with the same precedence will be resolved using the associativity: if the token is left-associative, then the parser will reduce; if the token is right-associative, then the parser will shift. If the token is non-associative, then the parser will declare a syntax error.
- When a shift/reduce conflict cannot be resolved using the above method, then ocamlyacc will output a warning and the parser will always shift.

yacc: o token `error`

o que acontece se o yacc detecta um erro (sintáctico) durante a análise sintáctica?

isto acontece quando a palavra actualmente analisada deixa de poder ter continuação que permita ser reconhecida

neste caso ferramentas como *yacc* utilizam a técnica da **correcção local do erro**

esta técnica baseia-se na introdução dum token especial, o token *error*, no local onde o erro foi detectado e na presença de tokens que desempenham um papel de **sincronização**

esses tokens são em geral tokens de pontuação que permitam delimitar componentes importantes da linguagem em causa

o token *error* pode assim ser utilizado como um token normal na gramática. A sua utilização num local particular da gramática representa o facto que a produção em questão pretende tratar um eventual erro sintáctico onde o token *error* aparece

que faz o yacc com o símbolo *error* introduzido aquando da detecção efectiva dum erro?

1. esvaziar a pilha de estados (se necessário) até chegar a um estado no qual uma acção para o token *error* seja *shift*
2. executar *shift error*
3. descartar símbolos de entrada (se necessário) até atingir um estado que tenha uma acção normal (que não seja de "erro") sobre o token de antevisão (**lookahead token**). Este token é o **token de sincronização**
4. retomar o *parsing* normalmente.

os tokens de sincronização podem ser pontuação que separam instruções

neste caso se um erro ocorrer, vamos procurar o próximo ponto de sincronização e ignorar todo o input que ocorre entre o local de erro e este ponto

a análise pode recomeçar normalmente a seguir. I.e. deita fora toda a instrução onde o erro ocorreu e retoma a analise normalmente com a instrução seguinte

(Documentação do OCamlYacc)

- Error recovery is supported as follows: when the parser reaches an error state (no grammar rules can apply), it calls a function named `parse_error` with the string "syntax error"as argument. The default `parse_error` function does nothing and returns, thus initiating error recovery (see below). The user can define a customized `parse_error` function in the header section of the grammar file.
- The parser also enters error recovery mode if one of the grammar actions raises the `Parsing.Parse_error` exception.
- In error recovery mode, the parser discards states from the stack until it reaches a place where the error token can be shifted. It then discards tokens from the input until it finds three successive tokens that can be accepted, and starts processing with the first of these. If no state can be uncovered where the error token can be shifted, then the parser aborts by raising the `Parsing.Parse_error` exception.

a ferramenta `menhir`

Menhir é um ferramenta que transforma uma gramática num analisador OCaml ; este assenta numa análise LR(1)

cada produção da gramática está acompanhada de uma **ação semântica** *i.e.* de código OCaml que constrói um valor semântico (tipicamente uma árvore de sintaxe abstracta)

Menhir utiliza-se em conjunto com um analisador léxico (tipicamente `ocamllex`)

um ficheiro Menhir tem o sufixo `.mly` e tem por estrutura

```
%{
  ...(opcional)  código OCaml arbitrário ...
%}
...declaração dos tokens...
...declaração de precedências e associatividade...
...declaração dos pontos de entrada...
%%
não-terminal-1:
| produção { ação }
| produção { ação }
;

não-terminal-2:
| produção { ação } ;
...
%%
  ... (opcional)  código OCaml arbitrário ...
```

```
%token PLUS LPAR RPAR EOF
%token <int> INT
%start <int> frase
%%

frase:
  e = expression; EOF  { e }
;

expression:
| e1 = expression; PLUS; e2 = expression { e1 + e2 }
| LPAR; e = expression; RPAR              { e }
| i = INT                                 { i }
;
```

compilamos o ficheiro `arith.mly` da forma seguinte

```
% menhir -v arith.mly
```

obtemos código OCaml puro no ficheiro `arith.ml(i)`, que contem em particular

- a declaração de um tipo `token`

```
type token = RPAR | PLUS | LPAR | INT of int | EOF
```

- para cada não-terminal declarado com `%start`, uma função de tipo

```
val frase: (Lexing.lexbuf -> token) -> Lexing.lexbuf -> int
```

como vemos, esta função toma como parâmetro um analisador léxico, do tipo dos que `ocamllex` produz (cf aula sobre análise léxica)

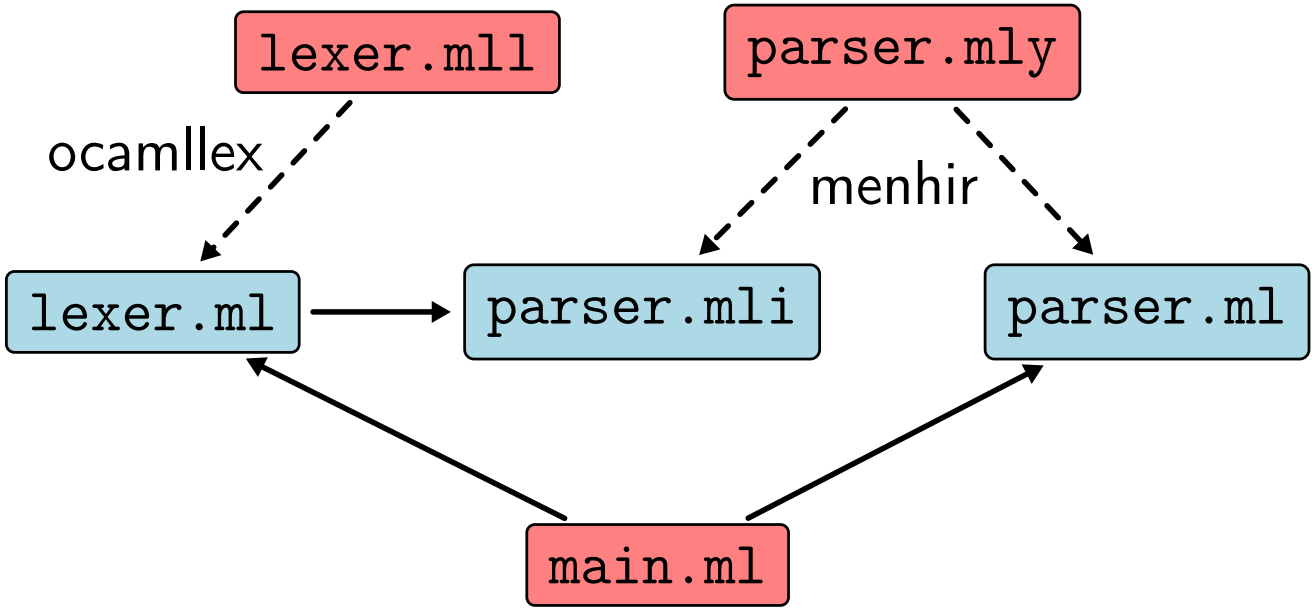
quando combinamos `ocamllex` e `menhir`

- `lexer.mll` faz referência aos lexemas definidos em `parser.mly`

```
{
  open Parser
}
...
```

- o analisador léxico e o analisador sintáctico são combinados da forma seguinte:

```
let c  = open_in file in
let lb = Lexing.from_file c in
let e  = Parser.frase Lexer.token lb in
...
```



= fonte utilizador

= construido automaticamente

= dependência

revisitemos a noção de conflito, mas desta vez com o Menhir

quando a gramática não é LR(1), Menhir apresenta os **conflitos** ao utilizador

- o ficheiro `.automaton` contém uma descrição do autómato LR(1) ; os conflito estão ali referenciados
- o ficheiro `.conflicts` contém, se assim for o caso, uma explicação de cada conflito, na forma de uma sequência de lexemas que conduz a duas árvores de derivação (i.e. a uma manifestação do conflito)

na gramática anterior, Menhir assinala o conflito

```
% menhir -v arith.mly
Warning: one state has shift/reduce conflicts.
Warning: one shift/reduce conflict was arbitrarily resolved.
```

o ficheiro `arith.automaton` contém em particular

```
State 6:
expression -> expression . PLUS expression [ RPAR PLUS EOF ]
expression -> expression PLUS expression . [ RPAR PLUS EOF ]
- On PLUS shift to state 5
- On RPAR reduce production expression -> expression PLUS expression
- On PLUS reduce production expression -> expression PLUS expression
- On EOF reduce production expression -> expression PLUS expression
** Conflict on PLUS
```

o ficheiro `arith.conflicts` apresenta a seguinte explicação do conflito

```
** Conflict (shift/reduce) in state 6.
** Token involved: PLUS
** This state is reached from phrase after reading:

expression PLUS expression

** In state 6, looking ahead at PLUS, shifting is permitted
** because of the following sub-derivation:

expression PLUS expression
                  expression . PLUS expression

** In state 6, looking ahead at PLUS, reducing production
** expression -> expression PLUS expression
** is permitted because of the following sub-derivation:

expression PLUS expression // lookahead token appears
```

uma forma de resolução dos conflitos passa por indicar ao Menhir como escolher entre leituras e reduções

para esse efeito, podemos dar **prioridades** aos lexemas e às produções e regras de **associatividades**

como no caso do yacc

por omissão, a prioridades de uma produção é a prioridade do seu lexema mais a direita (mas esta pode ser especificada de forma explícita)

se a prioridade da produção é maior do que o lexema por ler,
então favorece-se a redução

reciprocamente, sa prioridade do lexema for maior,
então a leitura é privilegiada

em caso de igualdade, consulta-se as regras de associatividade : um lexema associativo à esquerda favorece a redução e um lexema associativo à direita favorece a leitura

no nosso exemplo, basta indicar por exemplo que PLUS é associativo à esquerda

```
%token PLUS LPAR RPAR EOF
%token <int> INT
%left PLUS
%start <int> frase
%%
frase:
  e = expression; EOF  { e }
;
expression:
| e1 = expression; PLUS; e2 = expression { e1 + e2 }
| LPAR; e = expression; RPAR              { e }
| i = INT                                 { i }
;
```

para associar prioridades aos lexemas, utilizamos a convenção seguinte :

- a ordem de declaração da associatividade fixa as prioridades
  (os primeiros lexemas têm as prioridades mais fracas)
- vários lexemas podem aparecer numa mesma declaração (na mesma linha),
  tendo assim a mesma prioridade

exemplo :

```
%left PLUS MINUS
%left TIMES DIV
```

a gramática contém um conflito

```
expression:
| IF e1 = expression; THEN; e2 = expression
    { ... }
| IF e1 = expression; THEN; e2 = expression;
  ELSE; e3 = expression
    { ... }
| i = INT
    { ... }
| ...
```

corresponde à situação

```
IF a THEN IF b THEN c ELSE d
```

para associar o ELSE ao THEN mais próximo, devemos privilegiar a leitura

```
%nonassoc THEN
%nonassoc ELSE
```

(esta situação é conhecida em inglês como *dangling else*)

Menhir oferece numerosas vantagens relativamente a ferramentas como, por exemplo, `ocamlyacc` :

- não-terminais parametrizados por (não-)terminais
  - em particular, mecanismos para escrever expressões regulares na gramática ($E?$, $E*$, $E+$), listas com separadores

```
expr: (...);
lista_exprs:  ve = separated_list(COMMA,expr)  { ve } ;
```

- explicações dos conflitos
- modo interactivo, modo depuração
- gramáticas parametrizadas
- análise LR(1) no lugar de LALR(1)
- etc.

ler o manual de Menhir !

para que as fases seguintes da análise sintáctica (e.g. a tipagem) possam dar indicações de **posição** nas mensagens de erro, convém guardar informação de localização directamente na árvore de sintaxe abstracta

Menhir fornece esta informação via `$startpos` e `$endpos`, dois valores de tipo `Lexing.position` ; esta informação é-lhe transmitida pelo analisador léxico

cuidado : relembramos que `ocamllex` só mantém por omissão a posição absoluta dentro do ficheiro ; ter os números de linha e coluna actualizados necessita de um tratamento explícito realizado pelo programador (quem define os analisadores), (ver por exemplo, os vários ficheiros `lexer.mll` fornecidos nas aulas práticas)

uma forma de conservar a informação de localização na AST é a seguinte (ver aula sobre tipagem)

```
type expression =
  { desc: desc;
    loc : Lexing.position * Lexing.position }

and desc =
  | Econst of int
  | Eplus  of expression * expression
  | Eneg   of expression
  | ...
```

a gramática pode então parecer-se com

```
expression:
| d = desc { { desc = d; loc = $startpos, $endpos } }
;

desc:
| i = INT                                    { Econst i }
| e1 = expression; PLUS; e2 = expression { Eplus (e1, e2) }
| ...
```

como no caso de `ocamllex`, é necessário assegurar a aplicação de `menhir` antes do cálculo das dependências

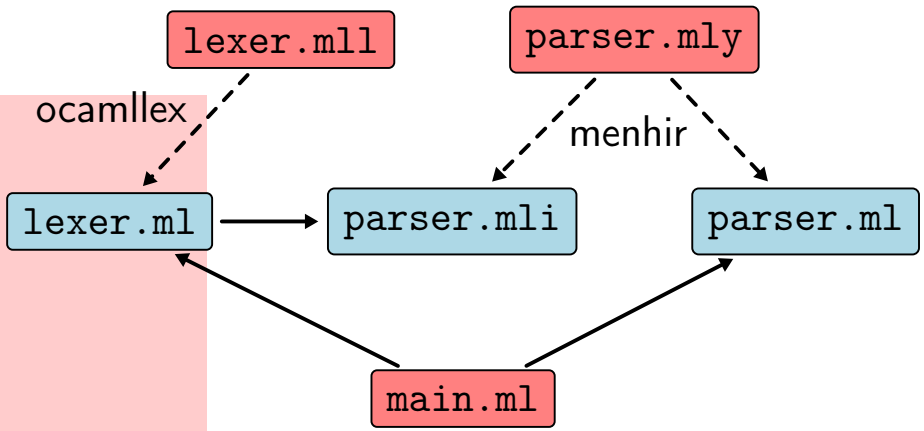
o `Makefile` pode ser algo como :

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

parser.mli parser.ml: parser.mly
        menhir -v parser.mly

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

include .depend
```



... ou então, utilizar `ocamlbuild`

conclusão

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)