Interpretation and Compilation of Languages

Master Programme in Computer Science

Mário Pereira   mjp.pereira@fct.unl.pt

Nova School of Science and Technology, Portugal

March 23, 2025

Lecture 3

based on lectures by Jean-Christophe Filliâtre and Léon Gondelman
previous editions by João Costa Seco, Luís Caires, and Bernardo Toninho

What do you see?

Heróisdomar,nobrepovo,Naçãovalente,imortal,Levantaihojedenovo
OesplendordePortugal!Entreasbrumasdamemória,ÓPátria,sente-seavoz
Dosteusegrégiosavós,Quehá-deguiar-teàvitória!

Heróis do mar, nobre povo,
Nação valente, imortal,
Levantai hoje de novo
O esplendor de Portugal!
Entre as brumas da memória,
Ó Pátria, sente-se a voz
Dos teus egrégios avós,
Que há-de guiar-te à vitória!

What do you see?

Asarmaseosbarõesassinalados Quedaa OcidentalpraiaLusitana,
PormaresnuncadantesnavegadosPassaramaindaalémdaTaprobana,
EmperigoseguerrasesforçadosMaisdoqueprometiaaforçahumana
EentregenteremotaedificaramNovoReino,quetantosublimaram;

What do you see?

As armas e os barões assinalados
Que da Ocidental praia Lusitana,
Por mares nunca dantes navegados
Passaram ainda além da Taprobana,
Em perigos e guerras esforçados
Mais do que prometia a força humana
E entre gente remota edificaram
Novo Reino, que tanto sublimaram;

Lexical analysis goal is to split the input into “words”.

As with natural languages, this splitting phase eases the work of the next step, the syntactic analysis.

The resulting words are called tokens.

Today: Lexical Analysis

1. Regular expressions
2. Finite automata
3. Lexical analyzer
    - Principles
    - Construction
4. Lexing tools
    - `ocamllex`
    - `jflex`
5. Lexers in practice

ITA Demo: using `ocamllex` for fun and profit

Example

source = sequence of characters

```
fun x -> (* my function *)
    x+1
```

↓

lexical analysis

↓

sequence of tokens

`fun` `x` `->` `x` `+` `1`

⋮

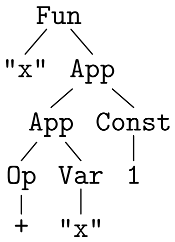
⋮

↓

syntax analysis

↓

abstract syntax



Blanks (spaces, newlines, tabs, etc.) play a role in lexical analysis; they can be used to separate two tokens.

For instance, `funx` is understood as a single token (identifier `funx`) and `fun x` is understood as two tokens (keyword `fun` and identifier `x`).

Yet several blanks are useless (as in `x + 1`)
and simply ignored.

Blanks do not appear in the returned sequence of tokens.

Lexical conventions differ according to the languages, and some blanks may be significant.

Examples:

- tabs for `make`
- newlines and indentation in Python or Haskell (indentation defines the structure of blocks)

Comments act as blanks

```
fun(* go! *)x -> x + (* adding one *) 1
```

Here the comment `(* go! *)` is a significant blank (splits two tokens) and the comment `(* adding one *)` is a useless blank.

Note: comments are sometimes interpreted by other tools (`javadoc`, `ocamldoc`, etc.), which handle them differently in their own lexical analysis

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

Which tools?

To implement lexical analysis, we are going to use

- regular expressions to describe tokens
- finite automata to recognize them

We exploit the ability to automatically construct a deterministic finite automaton recognizing the language described by a regular expression.

Regular Expressions

Let A be some alphabet

$$
\begin{array}{rcll}
r & ::= & \emptyset & \text{empty language} \\
  & | & \epsilon & \text{empty word} \\
  & | & a & \text{character } a \in A \\
  & | & r\,r & \text{concatenation} \\
  & | & r \,|\, r & \text{alternation} \\
  & | & r\star & \text{Kleene star}
\end{array}
$$

Conventions: in forthcoming examples, star has strongest priority, then concatenation, then alternation e.g. $r_0\,r_1 \,|\, r_2\,r_3\star$ means $(r_0\,r_1) \,|\, (r_2\,(r_3)\star)$

The language defined by the regular expression r is the set of words $L(r)$ defined as follows:

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

With alphabet $\{a, b\}$

- words with exactly three letters

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

- words ending with a

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

- words alternating a and b

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

indeed, we have $\epsilon = \epsilon(ab)^0\epsilon$, $a = \epsilon(ab)^0 a$, $b = b(ab)^0\epsilon$ and

$a \ldots a$	$a \ldots b$	$b \ldots b$	$b \ldots a$
	$ab = \epsilon(ab)^1\epsilon$		$ba = b(ab)^0 a$
$aba = \epsilon(ab)^1 a$	$abab = \epsilon(ab)^2\epsilon$	$bab = b(ab)^1\epsilon$	$baba = b(ab)^1 a$
$ab(ab)^n a = \epsilon(ab)^{n+1} a$	$a(ba)^n b = \epsilon(ab)^{n+1}\epsilon$	$b(ab)^n ab = b(ab)^{n+1}\epsilon$	$b(ab)^n a = b(ab)^{n+1} a$

Decimal integer literals, possibly with leading zeros

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

Identifiers composed of letters, digits and underscore, starting with a letter

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

Floating point numbers (`3.14` `2.` `1e-12` `6.02e23` etc.)

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

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

Comments such as `(* ... *)`, `(** ... *)` but not nested (that is, not `(* ... (* ... *) ... *)`), can be described with the following regular expression

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

where $r_1 =$ all characters but $*$ and $)$
and $r_2 =$ all characters but $*$

Regular expressions are not expressive enough to describe nested comments (we say that the language of balanced parentheses is not regular).

We will explain later how to get around this problem.

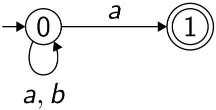
Finite Automata

Definition

A finite automaton over some A is a tuple (Q, T, I, F) where

- *Q is a finite set of states*
- *$T \subseteq Q \times A \times Q$ is a set of transitions*
- *$I \subseteq Q$ is a set of initial states*
- *$F \subseteq Q$ is a set of final states*

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



A word $a_1 a_2 \dots a_n \in A^\star$ is recognized by the automaton (Q, T, I, F) if and only if

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

with $s_0 \in I$, $(s_{i-1}, a_i, s_i) \in T$ for all i, and $s_n \in F$.

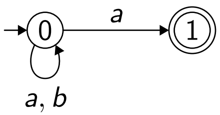
The language defined by an automaton is the set of words it recognizes.

Result

Theorem (Kleene, 1951)

Regular expressions and finite automata define the same languages.

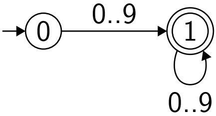
$(a|b)\star a$



Regular expression

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

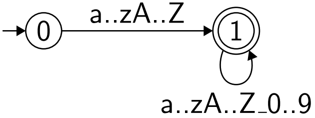
Automaton



Regular expression

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

Automaton

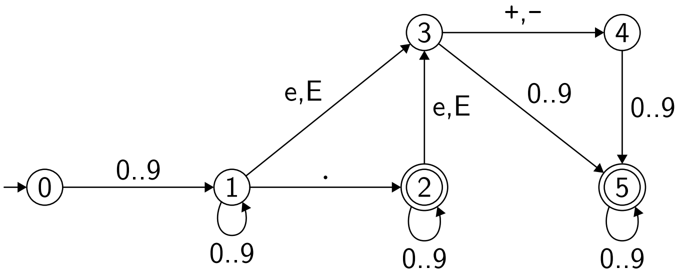


Regular expression

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

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

Automaton

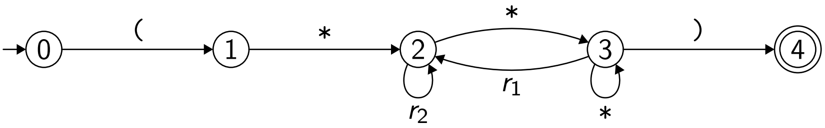


Regular expression

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

where r_1 = all characters but $*$ and)
and r_2 = all characters but $*$

Automaton



Lexical Analyzer

A lexical analyzer is a finite automaton for the "union" of all regular expressions describing the tokens.

However, it differs from the mere analysis of a single word by an automaton, since

- we must split the input into a sequence of words
- there are possible ambiguities
- we have to build tokens (final states contain actions)

The word `funx` is recognized by the regular expression for identifiers, but contains a prefix recognized by another regular expression (keyword `fun`)

⇒ we choose to match the longest token

The word `fun` is recognized by the regular expression for the keyword `fun` but also by that of identifiers

⇒ we order regular expressions using priorities

No Backtracking

With the three regular expressions

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

a lexical analyzer will fail on input

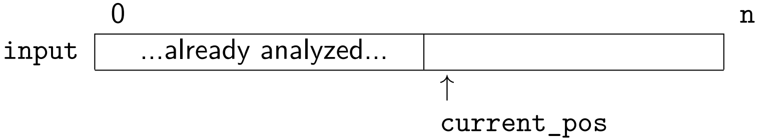
$$abc$$

(ab is recognized, as longest, then failure on c)

Yet the word abc belongs to the language $(a|ab|bc)\star$

Tokens are output one by one, on demand (from the syntax analyzer).

The lexical analyzer memorizes the position where the analysis will resume.



When a new token is required, we start from the initial state of the automaton, from position `current_pos`.

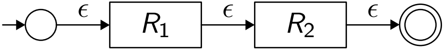
As long as a transition exists, we follow it, while memorizing any token that was recognized (any final state that was reached).

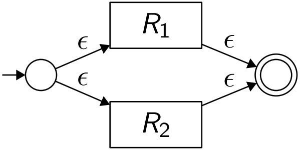
When there is no transition anymore, there are two cases:

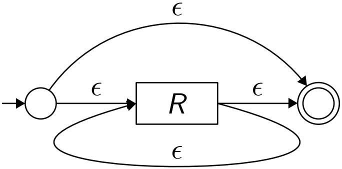
- if a token was recognized, we return it and `current_pos`$\leftarrow$`last`
- otherwise, we signal a lexical error

Building the Automaton

We can build a finite automaton corresponding to a regular expression by means of an intermediate non-deterministic automaton.







We can then convert it into a deterministic automaton, even minimize it

but. . .

... we can also build directly a deterministic automaton.

Starting idea: we can match characters from a recognized word and those from and those appearing in a regular expression.

Example: the word *aabaab* is recognized by $(a|b)\star a(a|b)$, as follows

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

Let us distinguish the different characters of the regular expression:

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

We build an automaton where the states are sets of characters.

The state s recognizes suffixes of $L(r)$ for which the first character belongs to s.

Example: the state $\{a_1, a_2, b_1\}$ recognizes words where the first is either an a matching a_1 or a_2, or a b matching b_1.

To build transitions from s_1 to s_2, one needs to compute the characters that can occur after another in a recognized word (*follow*).

Example: still using $r = (a_1|b_1) \star a_2(a_3|b_2)$, we have

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

To compute *follow*, we need to compute the first (resp. last) possible characters in a recognized word (*first*, resp. *last*)

Example : still using $r = (a_1|b_1) \star a_2(a_3|b_2)$, we have

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

To compute *first* and *last*, we need a last notion : does the empty word belongs to the recognized language ? (*null*)

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

We can thus define *first*...

$$
\begin{array}{rcl}
\mathit{first}(\emptyset) & = & \emptyset \\
\mathit{first}(\epsilon) & = & \emptyset \\
\mathit{first}(a) & = & \{a\} \\
\mathit{first}(r_1\, r_2) & = & \mathit{first}(r_1) \cup \mathit{first}(r_2) \quad \text{if } \mathit{null}(r_1) \\
 & = & \mathit{first}(r_1) \quad \text{otherwise} \\
\mathit{first}(r_1 \,|\, r_2) & = & \mathit{first}(r_1) \cup \mathit{first}(r_2) \\
\mathit{first}(r\star) & = & \mathit{first}(r)
\end{array}
$$

... and *last*

$$\begin{array}{rcl}
last(\emptyset) & = & \emptyset \\
last(\epsilon) & = & \emptyset \\
last(a) & = & \{a\} \\
last(r_1\, r_2) & = & last(r_1) \cup last(r_2) \quad \text{if } null(r_2) \\
 & = & last(r_2) \quad \text{otherwise} \\
last(r_1 \,|\, r_2) & = & last(r_1) \cup last(r_2) \\
last(r\star) & = & last(r)
\end{array}$$

And finally we can define *follow*

$$
\begin{array}{rcl}
\mathit{follow}(c, \emptyset) & = & \emptyset \\
\mathit{follow}(c, \epsilon) & = & \emptyset \\
\mathit{follow}(c, a) & = & \emptyset \\
\mathit{follow}(c, r_1\, r_2) & = & \mathit{follow}(c, r_1) \cup \mathit{follow}(c, r_2) \cup \mathit{first}(r_2) \quad \text{if } c \in \mathit{last}(r_1) \\
 & = & \mathit{follow}(c, r_1) \cup \mathit{follow}(c, r_2) \quad \text{otherwise} \\
\mathit{follow}(c, r_1 \,|\, r_2) & = & \mathit{follow}(c, r_1) \cup \mathit{follow}(c, r_2) \\
\mathit{follow}(c, r\star) & = & \mathit{follow}(c, r) \cup \mathit{first}(r) \quad \text{if } c \in \mathit{last}(r) \\
 & = & \mathit{follow}(c, r) \quad \text{otherwise}
\end{array}
$$

We have everything we need to build the automaton for regular expression s.

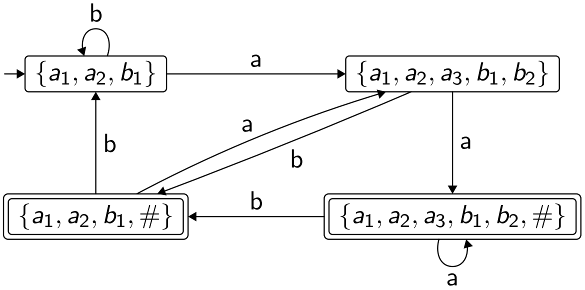
We begin by adding a character $\#$ (*End of File*) to the end of r.
The algorithm is as follows:

1. the initial state is the set $\mathit{first}(r\#)$

2. while there is a state s for which one needs to compute transitions
   for each character c of the alphabet
   let s' be the state $\bigcup_{c_i \in s} \mathit{follow}(c_i, r\#)$
   add the transition $s \xrightarrow{c} s'$

3. The accepting states are those that contain $\#$

For $(a|b)\star a(a|b)$ we get



Reference : Gérard Berry & Ravi Sethi,
From regular expressions to deterministic automata, 1986

Lexing Tools

In practice, we have tools to build lexical analyzers from a description with regular expressions and actions.

This is the `lex` family: `lex`, `flex`, `jflex`, `ocamllex`, etc.

We illustrate `jflex` (for JAVA) and `ocamllex` (for OCAML).

To illustrate these tools, let us write a lexical analyzer for a language of arithmetic expressions with

- integer literals
- parentheses
- subtraction

jflex

A `jflex` file has suffix `.flex` and the following structure

```
... preamble ...
%{
... some Java code
%}
%%
<YYINITIAL> {
  regular expression { action }
  ...
  regular expression { action }
}
```

where each `action` is Java code
(returning a token most of the time).

Example (1/2)

We set up a file `Lexer.flex` for our language

```
import static sym.*;  /* imports the tokens */

%%

%class Lexer              /* our class will be Lexer */
%unicode                  /* we use unicode characters */
%cup                      /* syntax analysis using cup */
%line                     /* activate line numbers  */
%column                   /* and column numbers     */
%yylexthrow Exception /* we can raise Exception */

%{
   /* no need for a Java preamble here */
%}
```

...

...

```
WhiteSpace          = [ \t\r\n]+    /* shortcuts */
Integer             = [:digit:]+
%%
<YYINITIAL> {
  "-"   { return new Symbol(MINUS, yyline, yycolumn); }
  "("   { return new Symbol(LPAR, yyline, yycolumn); }
  ")"   { return new Symbol(RPAR, yyline, yycolumn); }
  {Integer}
        { return new Symbol(INT, yyline, yycolumn,
                             Integer.parseInt(yytext())); }
  {WhiteSpace}
        { /* ignore */ }
  .     { throw new Exception (String.format (
           "Line %d, column %d: illegal character: '%s'\n",
           yyline, yycolumn, yytext())); }
}
```

Explaining

- tokens are freely implemented;
  here, we use the class `Symbol` that comes with cup[1]

- `MINUS`, `LPAR`, `RPAR` and `INT` are integers (token kinds)
  built by the tool cup and imported from `sym.java`

- variables `yyline` and `yycolumn` are updated automatically

- `yytext()` returns the string that was recognized by the regular expression

[1] the JAVA syntactic analyzer

We compile file `Lexer.flex` with `jflex`

```
jflex Lexer.flex
```

We get pure Java code in `Lexer.java`, with

- a constructor

```
Lexer(java.io.Reader)
```

- a method

```
Symbol next_token()
```

`jflex` Regular Expressions – Summary

`.`	any character
`a`	the character `'a'`
`"foobar"`	the string `"foobar"` (in particular $\epsilon =$ `""`)
`[`*characters*`]`	set of characters (e.g. `[a-zA-Z]`)
`[^`*characters*`]`	set complement (e.g. `[^"]`)
`[:`*ident*`:]`	predefined set of characters (e.g. `[:digit:]`)
`{`*ident*`}`	named regular expression
r_1 `|` r_2	alternation
$r_1\,r_2$	concatenation
r `*`	star
r `+`	one or more repetitions of r ($\stackrel{\text{def}}{=} r\,r\star$)
r `?`	zero or one occurrence of r ($\stackrel{\text{def}}{=} \epsilon \mid r$)
`(` r `)`	grouping

ocamllex

An `ocamllex` file has suffix `.mll` and the following structure

```
{
  ... some OCaml code ...
}
rule ident = parse
| regular expression { action }
| regular expression { action }
| ...
{
  ... some OCaml code ...
}
```

where each `action` is some OCaml code.

```
let white_space = [' ' '\t' '\n']+
let integer     = ['0'-'9']+
rule next_token = parse
  | white_space
      { next_token lexbuf }
  | integer as s
      { INT (int_of_string s) }
  | '-'
      { MINUS }
  | '('
      { LPAR }
  | ')'
      { RPAR }
  | eof
      { EOF }
  | _ as c
      { failwith ("illegal character" ^ String.make 1 c) }
```

- we assume the following type for the tokens

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

  (will be built by the syntax analyzer)

- contrary to `jflex`
  - we explicitly call `next_token` to ignore blanks
  - we do not handle lines and columns explicitly

We compile `lexer.mll` with `ocamllex`

```
% ocamllex lexer.mll
```

that generates OCaml `lexer.ml` file that defines a function for each analyzer `f1, ..., fn` :

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

and

```
val next_token : Lexing.lexbuf -> token
```

The type `Lexing.lexbuf` is the one of the data structure for the state of the lexical analyzer.

The module `Lexing` of the standard library gives different ways to construct a value of this type such as

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

```
val from_string : string -> lexbuf
```

`_`	any character
`'a'`	the character `'a'`
`"foobar"`	the string `"foobar"` (in particular $\epsilon =$ `""`)
`[`*characters*`]`	set of characters (e.g. `['a'-'z' 'A'-'Z']`)
`[^`*characters*`]`	set complement (e.g. `[^ '"']`)
ident	named regular expression
r_1 `|` r_2	alternation
$r_1\, r_2$	concatenation
r `*`	star
r `+`	one or more repetitions of r ($\stackrel{\text{def}}{=} r\, r\star$)
r `?`	zero or one occurrence of r ($\stackrel{\text{def}}{=} \epsilon \mid r$)
`(` r `)`	grouping
`eof`	end of input

Identifiers

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

Integer constants

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

Floating number constants

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

```
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)  { ... }
```

When using the keyword parse, it is a rule of the longest recognized token that is applied.

And for two recognized tokens of equal length, it's the first rule that applies

```
| "fun"              { Tfun }
| ['a'-'z']+ as s   { Tident s }
```

For the shortest, one can use the keyword shortest instead of parse

```
rule scan = shortest
  | regexp1 { action1 }
  | regexp2 { action2 }
  ...
```

One can give a name to a recognized token or a sub-token using as construct and then use it inside the action

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

| (['+' '-']? as sign) (['0'-'9']+ as num)  { action }
```

Inside of an action, it is possible to call the lexical analyzer recursively, or call other mutually recursively defined rules

The buffer of the lexical analyzer must be passed as argument; it is contained inside a variable called `lexbuf`

For example; it's easy to treat white spaces:

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

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

and comment = parse
  | "*)"  { token lexbuf }
  | _     { comment lexbuf }
  | eof   { failwith "non terminated comment!" }
```

Note how we handle properly an error of non-terminated comment

We can also handle nested comments using a lexical analyzer.

First try: using a counter. . .

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

and comment level = parse
  | "*)"  { if level > 1 then comment (level - 1) lexbuf }
  | "(*"  { comment (level + 1) lexbuf }
  | _     { comment level lexbuf }
  | eof   { failwith "non terminated comment" }
```

... or even without a counter!

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

and comment = parse
  | "*)"  { () }
  | "(*"  { comment lexbuf; comment lexbuf }
  | _     { comment lexbuf }
  | eof   { failwith 'non terminated comment" }
```

Note: by doing so we actually go beyond regular expressions!

Four Rules to Follow

Four rules to keep in mind when writing a lexical analyzer:

1. handle white spaces
2. the rules with highest priory to define first (e.g. language keywords before identifiers)
3. signal lexical errors (e.g. illegal characters, non terminated comments, etc.)
4. handle the end of the input (`eof`)

Recap

- Regular expressions are the basis of lexical analysis
- The job is automatized with tools such as `jflex` and `ocamllex`
- `jflex`/`ocamllex` are more expressive than regular expressions

  indeed, actions can call the lexical analyzer recursively
  $\Rightarrow$ allows us to recognize nested comments for instance

It's Demo Time!

The use of `ocamllex` is not even limited to *classical* lexical analysis.

To analyze a text (string, file, stream) based on regular expressions, `ocamllex` is the tool of choice.

Nice example: write filters, *i.e.*, programs that translate a language into another, following local and simple modifications.

Example 1: counting the occurrences of a word in a text.

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

Example 2: remove blanks from a text.

that's right, I used `ocamllex` to build the first two slides ;)

```
{
  let buffer = Buffer.create 1024
}
let blank = ['\t' '\r' '\n' ' ']

rule token = parse
  | blank+ { token lexbuf }
  | _ as c { Buffer.add_char buffer c; token lexbuf }
  | eof    { () }
{
  let () =
    let fin = Sys.argv.(1) in
    let cin = open_in fin in
    token (Lexing.from_channel cin);
    let cout = open_out (fin ^ "_out") in
    Buffer.output_buffer cout buffer
}
```

Example 3: a small translator from WHILE language into HTML to prettify the online source code.

Goal

- usage: `while2html file.wl`, that outputs `file.wl.html`
- keywords in green, comments in red
- numbering lines

- Compilers, Principles, Techniques, and Tools, Alfred Aho and Monica S. Lam and Ravi Sethi and Jeffrey D. Ullman, Second Edition[Chapter 3], Addison-Wesley (2006) (*"the dragon book"*).