Compiler

Lexical analysis

Jyun-Ao Lin

iFIRST & CSIE, NTUT

Credits

A large part of this course is based on the Compilation Course of J.-C. Filliâtre at ENS Ulm.

Lexical analysis

The lexical analysis splits the input into "words/strings"

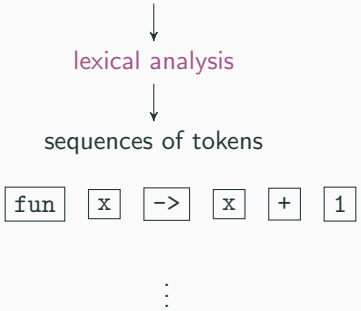
Same as in natural languages, splitting in words makes the later phase, ie, parsing (syntactic analysis), easier

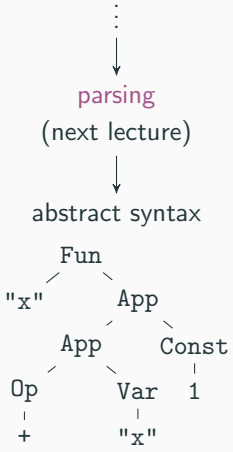
These words are called tokens

Lexical analysis: example

input = sequence of characters

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





Blanks

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

Blanks

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

Example

- tabs for `make`
- newlines and indentation in Python or Haskell
  (indentation defines the structure of blocks)
- carriage returns sometimes transformed into semicolons (Go, Julia, etc.)

Comments

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 tool?

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

Overview of the course

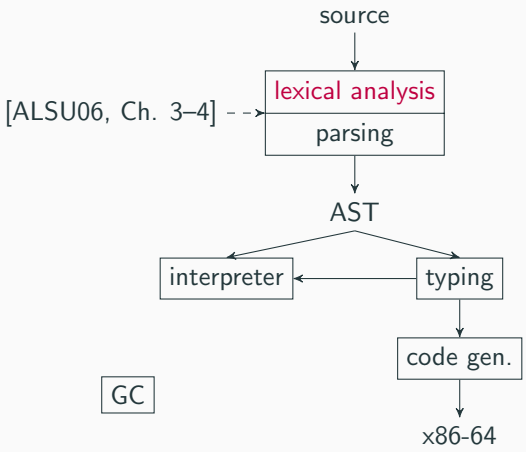


Table of contents

Regular expression

Syntax

Let Σ be some alphabet

$$
\begin{array}{rcll}
r & ::= & \emptyset & \text{empty language} \\
 & | & \epsilon & \text{empty string} \\
 & | & a & \text{character/symbol } a \in \Sigma \\
 & | & r\ r & \text{concatenation} \\
 & | & r \mid r & \text{alternation} \\
 & | & r* & \text{Kleene star}
\end{array}
$$

Conventions: in forthcoming examples, star has strongest priority, then concatenation, then alternation.

Semantics

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

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

Example

Example
With alphabet $\Sigma = \{a, b\}$

- words with exactly three letters

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

- words ending with a

$$(a|b) * a$$

- words alternating a and b

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

Integer literals

decimal integer literals, possibly with leading zeros

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

Identifiers

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

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

Floating point literals

floating point numbers in OCaml (`3.14 2. 1e-12 6.02e23` etc)

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

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

Comments

comments such as `(* ... *)`, not nested, can be described with the following regular expression

$$\boxed{(}\,\boxed{*}\,(\,\boxed{*}\,*\;\; r_1 \mid r_2 \;\;)\,*\,\boxed{*}\,\boxed{*}\,*\,\boxed{)}$$

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

Nested comments

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

Recreation

Let us give the type

```
type regexp =
  | Empty
  | Epsilon
  | Char     of char
  | Union    of regexp * regexp
  | Concat   of regexp * regexp
  | Star     of regexp
```

Write a function

```
val accepts: regexp -> char list -> bool
```

as simple as possible

Nullity

Let r be a regular expression; does the empty word ϵ belongs to $L(r)$?

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

Residue (Brzozowski derivation)

For a regular expression r and a character c, we define

$$\delta(r, c) = \{w \mid cw \in L(r)\}$$

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

End of the recreation

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

(20 lines)

Finite automata

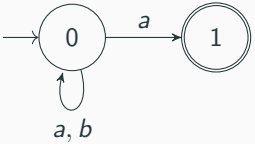
Finite automata

Defn. (finite automation)
A finite automation over some Σ is a tuple (Q, T, I, F) where

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

□

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



Language

Defn. (recognized words)
A word $w = a_1 a_2 \dots a_n \in \Sigma^*$ is recognized/accepted by the automation (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 $s_n \in F$.

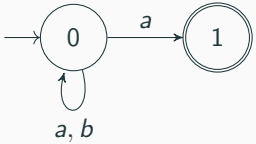
The language defined by an automation is the set of words it recognizes □

Result

Thm. (Kleene, 1951)
Regular expressions and finite automata define the same class of languages, called regular languages □

Example

```
(a|b)*a
```



Recognizing tokens

token	regular expression	automata
keyword `fun`	`f u n`	
symbol +	+	
symbol ->	- >	

Integer literals

regular expression

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

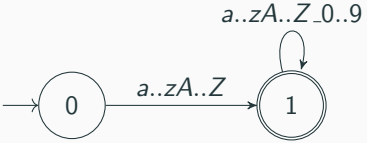
automata

Identifiers

regular expression

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

automata



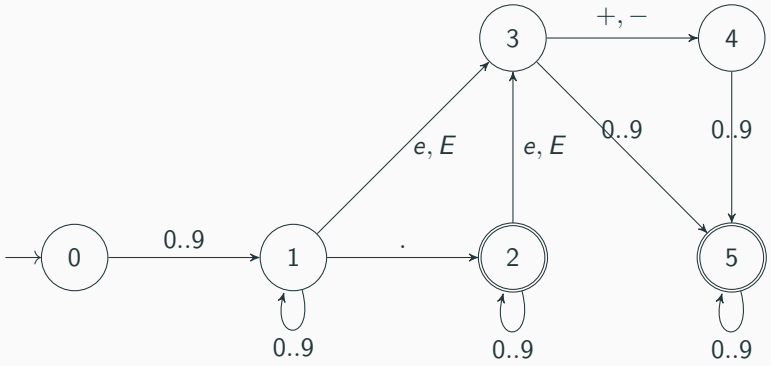
Floating point literals

regular expression

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

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

automata



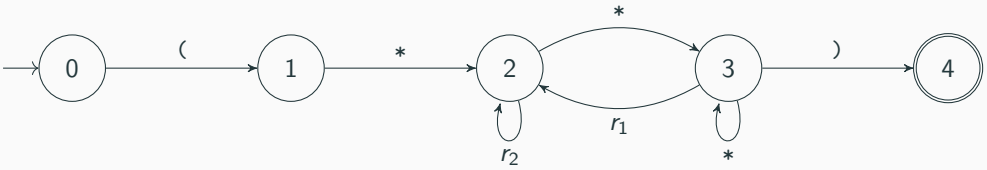
Comments

regular expression

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

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

automata



Lexical analysis

Lexical analyzer

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

However, in terms of functionality, 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 some actions)

Ambiguities

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

$\Longrightarrow$ 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

$\Longrightarrow$ 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 $(a|ab|bc)*$

Lexical analyzer

The lexical analyzer must remember the last final state encountered, if applicable

When there is no longer any possible transition in the automaton, one of two things happens

- no final state was stored $\Longrightarrow$ lexical analyzer fails
- it has read the prefix wv of the input, with w the token recognized by the last encountered final state $\Longrightarrow$ we return the token w, and the analyzer restart with v prefixed to the rest of the input.

A bit of coding

Let's code a lexical analyzer

We introduce a type of deterministic finite automata

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

with

```
type action =
  | NoAction (* no action = not a final state *)
  | Action of string (* nature of token *)
```

and

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

A small lexical analyzer

The transition table is full for states (array) and empty for the characters (AVL)

We are given

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

A small lexical analyzer

The goal is to write an `analyzer` function that takes an automaton and a string to analyze, and returns a function to calculate the next token

i.e.

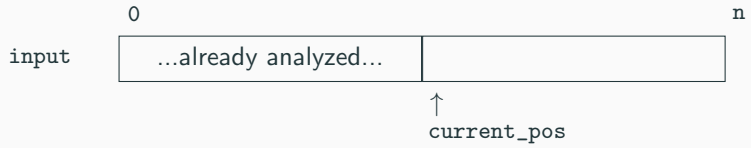
```
val analyzer: automaton -> string -> (unit -> string * string)
```

note: we could also return the list of tokens, but we adopt here the methodology which is used in practice in the interaction between lexical analysis and syntactic analysis

A small lexical analyzer

the lexical analyzer memorizes the current position in the input using a reference `current_pos`

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



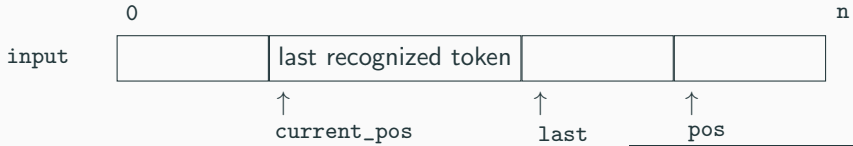
note: partial application of analyzer will be crucial

A small lexical analyzer

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

```
let analyzer autom input =
  let n = String.length input in
  let current_pos = ref 0 in
  fun () ->
    let rec scan last state pos =
    (* we are about to examine the character pos *) ...
    in
    scan None autom.initial !current_pos
```

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



A small lexical analyzer

we then determine whether a transition is possible

```
let rec scan last state pos =
  let state' =
    if pos = n then -1
    else transition autom state input.[pos]
  in
  if state >= 0 then
(* a transition to state' *) ...
  else
(* no possible transition *) ...
```

A small lexical analyzer

if yes, we update `last`, if necessary, and we call `scan` recursively on `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)
```

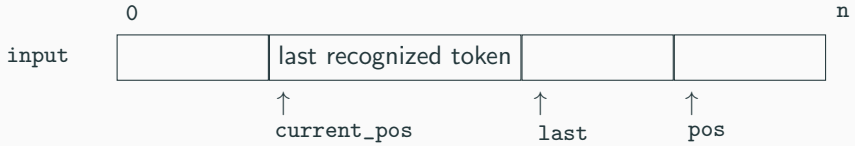
when `last` has the form `Some (p, a)`,

- `p` is the position following the recognized token
- `a` is the action (type of the token)

A small lexical analyzer

if on the contrary no transition is possible, we examine `last` to determine the result

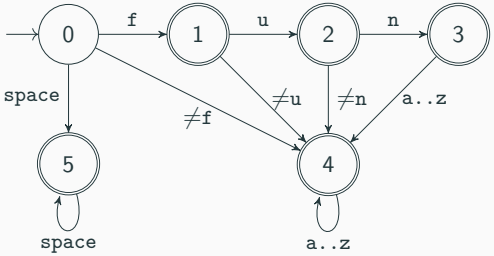
```
else match last with
  | None ->
      failwith "failure"
  | Some (last_pos, action) ->
      let start = !current_pos in
      current_pos := last_pos;
      action, String.sub input start (last_pos - start)
```



A small lexical analyzer

Let's test with

- a keyword: `fun`
- some identifiers: `(a..z)(a..z)*`
- some blanks: `space space*`



A small lexical analyzer

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

A small lexical analyzer

```
# 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 " failure".
```

Another possibility

There are of course other possibilities for programming a lexical analyzer

example: *n* mutually recursive functions, one per state of the automaton

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

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

Tools

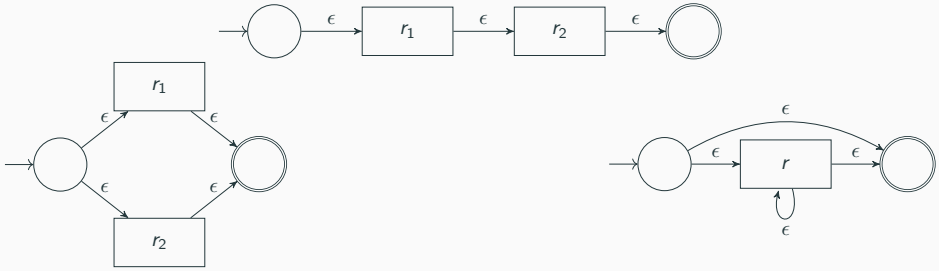
In practice, we have tools to build lexical analyzers from a description with regular expressions and actions

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

Construction of automation

Thompson algorithm

We can construct the finite state automata corresponding to a regular expression by passing through an intermediate nondeterministic automata



Then we can convert it to deterministic one, then minimize it.... but
(f.c. Formal Language course)

Direct construction (Berry & Sethi, 1986)

.... we can also directly construct the deterministic automata

Initial idea: match the characters of a recognized word and those appearing in the regular expression.

Example

The word *aabaab* belongs to the language of the regular expression $(a|b)*a(a|b)$ and we can make the following correspondence:

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

Direct construction

Distinguish the different characters of the regular expression, for example by associating them with distinct indices:

$$(a_1|b_1) * a_2(a_3|b_2)$$

We will construct an automaton whose states are sets of letters.

The state s recognizes words of $L(r)$ whose first letter belongs to s.

E.g., the state $\{a_1, a_2, b_1\}$ recognizes words whose first character is an a corresponding to a_1 or a_2 or is a b corresponding to b_1.

Transitions

To construct transitions between a state s_1 and a state s_2, it is necessary to determine the characters which can appear after another in a recognized word (*follow*)

Example
Always with $r = (a_1|b_1) * a_2(a_3|b_2)$, we have

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

First and last

To calculate *follow*, we need to calculate the first (resp. last) possible letters of a recognized word. (*first*, resp. *last*)

Example
Always with $r = (a_1|b_1) * a_2(a_3|b_2)$, we have

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

Nullity?

And to calculate *first* and *last*, we need one last notion: does the empty word belong to the language $L(r)$? ($null$)

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

(we have already done it before!)

Compute *first* and *last*

We can now define *first*

$$
\begin{aligned}
first(\emptyset) &= \emptyset \\
first(\epsilon) &= \emptyset \\
first(a) &= \{a\} \\
first(r_1 r_2) &= \begin{cases} first(r_1) \cup first(r_2) & \text{if } null(r_1) \\ first(r_1) & \text{otherwise} \end{cases} \\
first(r_1 | r_2) &= first(r_1) \cup first(r_2) \\
first(r*) &= first(r)
\end{aligned}
$$

Exercise: define *last* in a similar fashion.

Then the *follow*

Now we deduce the definition of *follow*, always by structural induction:

$$
\begin{aligned}
follow(c, \emptyset) &= \emptyset \\
follow(c, \epsilon) &= \emptyset \\
follow(c, a) &= \emptyset \\
follow(c, r_1 r_2) &= \begin{cases} follow(c, r_1) \cup follow(c, r_2) \cup first(r_2) & \text{if } c \in last(r_1) \\ follow(c, r_1) \cup follow(c, r_2) & \text{otherwise} \end{cases} \\
follow(c, r_1 | r_2) &= follow(c, r_1) \cup follow(c, r_2) \\
follow(c, r*) &= \begin{cases} follow(c, r) \cup first(r) & \text{if } c \in last(r) \\ follow(c, r) & \text{otherwise} \end{cases}
\end{aligned}
$$

Construction of the automata

Finally, we can construct the automata for the regular expression r

We start by adding at the end of r a character not belonging to the alphabet Σ; let us denote it by $\#$.

The algorithm is as follows:

1. The initial state is the set $first(r\#)$.
2. as long as there exists a state s for which the transitions must be calculated:
   for each character $c \in \Sigma$, let s' be the state $\bigcup_{c_i \in s} follow(c_i, r\#)$
   add the transition $s \overset{c}{s'}$
3. the final states are the states containing $\#$.

Construction of the automata

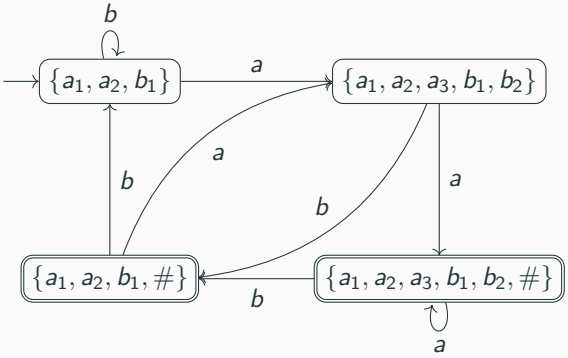
Algorithm 1 RegtoDFA(r)

Input `regex` r
Output DFA $M = (Q, T, I, F)$ with $L(M) = L(r)$

$Q, I \leftarrow first(r\#)$; $T, F \leftarrow \emptyset$
$W \leftarrow \{first(r\#)\}$;
while $W \neq \emptyset$ **do**
  pick $s \in W$; $W \leftarrow W \setminus \{s\}$
  for all $c \in \Sigma$ **do**
    Let $s' = \bigcup_{c_i \in s} follow(c_i, r\#)$
    $Q \leftarrow Q \cup \{s'\}$; $W \leftarrow W \cup \{s'\}$; $T \leftarrow T \cup \{(s, c, s')\}$
    if $\# \in s'$ **then**
      $F \leftarrow F \cup \{s'\}$
Return $M = (Q, T, I, F)$.

Example

Example
For $(a|b) * a(a|b)$, we obtain the following automaton:



Reference: Gérard Berry & Ravi Sethi, *From regular expressions to deterministic automata*, 1986 [BS86, Sec. 3–4]. See also [ALSU06, Sec. 3.9]

The tool `ocamllex`

Syntax

An ocamllex file has the suffix .mll and has the following form

```
{
    ... arbitrary OCaml code ...
}
rule f1 = parse
| regexp1 { action1 }
| regexp2 { action2 }
| ...
and f2 = parse
  ...
and fn = parse
  ...
{
    ... arbitrary OCaml code ...
}
```

The tool ocamllex

We compile the file `lexer.mll` with `ocamllex`

```
% ocamllex lexer.mll
```

which produces an OCaml `lexer.ml` file that defines a function for each parser `f1, ...,fn`

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

The type Lexing.lexbuf

The type Lexing.lexbuf is that of the data structure that contains the state of a lexical analyzer

The Lexing module of the standard library provides several ways to construct a value of this type, including

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

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

The regular expression of ocamllex

`_`	whatever character
`'a'`	the character `'a'`
`"foobar"`	the string `"foobar"` (in particular ϵ = `""`)
`[characters]`	the set of characters (e.g. `['a'-'z' 'A'-'Z']`)
`[^characters]`	complement (e.g. `[^ '"']`)
r_1 `|` r_2	alternative
$r_1 r_2$	concatenation
`r*`	Kleene star
`r+`	one or more repetitions of `r` ($\stackrel{\text{def}}{=}$ `r r*`)
`r?`	one or zero occurrence of `r` ($\stackrel{\text{def}}{=}$ ϵ `|` `r`)
`eof`	the end of the input

Examples

Identifiers

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

constant integers

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

constant floating

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

Shortcuts

We can define shortcuts for regular expressions

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

Ambiguities

For the analyzer defined with the keyword `parse`, the rule of longest recognized token applies

At equal length, the rule that appears first wins

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

For the shortest, just use `shortest` instead of `parse`

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

Recover the token

The string or substrings recognized by regular subexpressions can be named using the construct as

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

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

Treatment of blanks

In an action, it is possible to recursively call the lexical analyzer, or one of the other simultaneously defined analyzers

The lexical analysis buffer must be passed as an argument;
it is contained in a variable called `lexbuf`

It is thus very easy to deal with blanks:

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

Comments

To handle comments, we can use a regular expression

or a dedicated analyzer:

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

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

Advantage:
error related to an unclosed comment is correctly handled

Nested comments

Another interest: nested comments are easily handled with 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-closed comment" }
```

Nested comments

or without counter

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

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

Note: we have therefore exceeded the power of regular expressions

A complete example

Let us take the example of a micro language with integers, variables, `fun x -> e` and `e + e` constructions and comments of the form `(*...*)`

Given an OCaml type for the token

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

A complete example

```
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 ("illegal character : " ^ String.make 1 c) }
  | eof                 { Teof }

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

Four rules

Four rules to remember when writing a lexical analyzer

- deal with blanks (whitespace)
- the rules of highest priority first (e.g. keywords before identifiers)
- report lexical errors (illegal characters, but also unclosed comments or strings, illegal escape sequences, etc.)
- process end of input (eof)

Efficiency

By default, `ocamllex` constructs the automaton in a table, which is interpreted at runtime.

The `-ml` option makes it possible to produce pure OCaml code, where the automaton is encoded by functions; this is not recommended in practice however

Efficiency

Even using a table, the automaton can take up a lot of space, particularly if there are many key words in the language

It is preferable to use a single regular expression for identifiers and keywords, and then separate them using a table of keywords

```
{
    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 }
```

Case (in)sensibility

If you want a lexical analyzer that is not case sensitive, do not write

```
| ("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 }
| ...
```

but rather

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

compilation and dependencies

To compile (or recompile) OCaml modules, it is necessary to determine the dependencies between these modules and therefore ensure the prior production of the OCaml code with `ocamllex`

If we use `dune`, we indicate like this the presence of a file `lexer.mll` which must be processed with `ocamllex`

```
(ocamllex
  (modules lexer))

(executable
  (name minilang)
  ...
```

(c.f. the code provided in HW2 for example)

Applications of ocamllex

The use of ocamllex is not limited to lexical analysis

Whenever we want to analyze a text (string, file, stream) based on regular expressions, ocamllex is a tool of choice

In particular for writing filters, i.e. programs translating a language into another by local and relatively simple modifications

Example 1

Example
Combine several consecutive empty/blank lines into one

As simple as

```
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) }
```

We produce an executable with

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

and we use it like this

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

Example 2

Example
Count 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 3: caml2html

Example
A small OCaml to HTML translator, to beautify the source put online

Objective

- usage: `caml2html file.ml`, which produces `file.ml.html`
- keywords in green, comments in red
- number the lines
- all in less than 100 lines of code

caml2html

We write everything in a single file `caml2html.mll`

We begin by checking the command line

```
{
    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
```

then we open the HTML file in the scripture and we write in it with `fprintf`

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

caml2html

We write the beginning of the HTML file with the name of the file as title
we use the HTML tag <pre> to format the code

```
let () =
  print "<html><head><title>%s</title><style>" file;
  print ".keyword { color: green; } .comment { color: #990000; }";
  print "</style></head><body><pre>"
```

We introduce a function to number each line,
and we immediately invoke it for the first line

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

caml2html

We define a table of keywords (as for a lexical analyzer)

```
    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
}
```

We introduce a regular expression for identifiers

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

caml2html

We can attack the analyzer itself

For an identifier, we test if it is a keyword

```
rule scan = parse
  | ident as s
      { if is_keyword s then begin
          print "<span class=\"keyword\">%s</span>" s
        end else
          print "%s" s;
        scan lexbuf }
```

at each line break, we call the `newline` function:

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

caml2html

for a comment, we change color (red) and we invoke another comment analyzer; when it returns, we go back to the default color and we resume the scan analysis

```
| "(*"
    { print "<span class=\"comment\">(*";
        comment lexbuf;
        print "</span>";
        scan lexbuf }
```

any other character is printed as is

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

when it's over, it's over!

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

caml2html

for comments, just don't forget `newline`:

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

we finish with the application of `scan` on the input file

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

caml2html

this is almost correct:

- the character < is significant in HTML and must be written as `<`
- similarly you must escape & with `&`
- an OCaml string can contain `(*`
  (this code contains some `!`)

caml2html

Let's correct it

we add a rule for < and an analyzer for the strings

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

We have to pay attention to the character "", where " does not mark the beginning of a string

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

caml2html

We apply the same treatment in comments (small OCaml coquetry: we can comment on code containing "*)")

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

finally the strings are processed by the string analyzer, without forgetting the escape sequences (such as \" for example)

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

caml2html

Now it works correctly
(a good test is to try on `caml2html.mll` itself)

To do it properly, you should also convert the tabs at the beginning of the line (typically inserted by the editor) to spaces

left as an exercise...

Example 4

Example
Automatic indentation of programs C

Idea:

- with each opening brace, we increase the margin
- with each closing brace, we decrease it
- with each carriage return, we print the current margin
- without forgetting to process strings and comments

Automatic indentation of programs C

Given what we need to maintain the margin and display it

```
{
    open Printf

    let margin = ref 0
    let print_margin () =
      printf "\n%s" (String.make (2 * !margin) ' ')
}
```

Automatic indentation of programs C

At each carriage return, we ignore the spaces at the start of the line and we display the current margin

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

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

curly brackets change the current margin

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

Automatic indentation of programs C

small special case: a closing brace at the beginning of a line must decrease the margin before it is displayed

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

don't forget the character strings

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

nor the end of line comments

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

Automatic indentation of programs C

nor the comments of the form /* ... */

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

where

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

Automatic indentation of programs C

Finally, everything else is displayed as is, and `eof` finishes the analysis

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

The main program is two lines long

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

Automatic indentation of programs C

It's not perfect, of course

In particular, a loop or conditional body reduced to a single statement will not be indented:

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

Exercise: do the same thing for OCaml (this is much more difficult, and we can limit ourselves to a few simple cases)

Recap

- regular expressions are the basis of lexical analysis
- the work is largely automated by tools such as `ocamllex`
- `ocamllex` is more expressive than regular expressions, and can be used well beyond lexical analysis

Next

- HW3: compilation of regular expressions to finite automata
- Next lecture: parsing

References i


Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman.
Compilers: Principles, Techniques, and Tools (2nd Edition).
Addison-Wesley Longman Publishing Co., Inc., USA, 2006.


Gerard Berry and Ravi Sethi.
From regular expressions to deterministic automata.
Theoretical Computer Science, 48:117–126, 1986.

Questions?