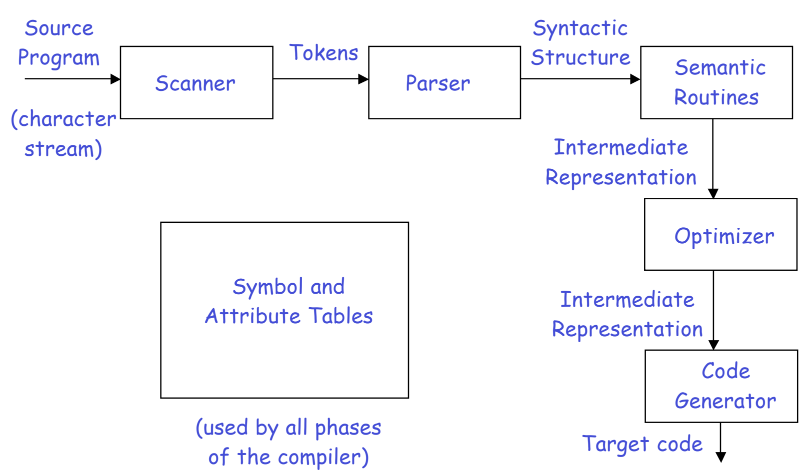
Lexical analysis

`ocamllex`

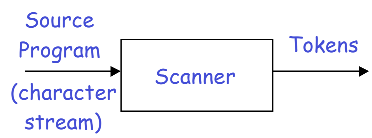
CS235 Languages and Automata

Department of Computer Science
Wellesley College

Compiler structure

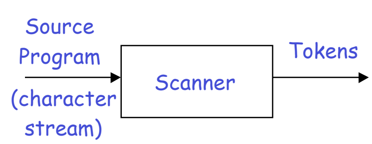


Lexical tokens



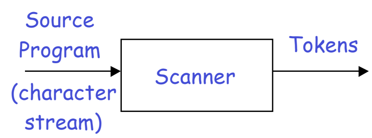
Type	Program Structure	Token Name
Keywords	**begin** **end** **write** **read**	BEGIN END WRITE READ
Identifier	Integer variable	ID
Literals	Integer constants	INTLIT
Operators	:= + -	ASSIGNOP PLUSOP MINUSOP
Punctuation	; ( ) ,	SEMICOLON LPAREN RPAREN COMMA

Nontokens



Type	Program Structure
Comments	`/* try again */`
Preprocessor directives	`#include<stdio.h>` `#define NUMS 5, 6`
macros	`NUMS`
blanks, tabs, and newlines	

Token stream



SLP source code

```
a := 5+3; b := (print(a, a-1), 10*a); print(b)
```

Token stream produced by scanner

```
ID(a)    ASSIGNSYM    NUM(5)    OPERATOR(+)    NUM(3)
SEMICOLON     ID(b)    ASSIGNSYM     LPAREN    PRINTSTM
LPAREN    ID(a)    COMMA    ID(a)    OPERATOR(-)    NUM(1)
RPAREN    COMMA    NUM(10)   OPERATOR(*)    ID(a)    RPAREN
SEMICOLON     PRINTSTM     LPAREN    ID(b)    RPAREN
```


How should lex rules be described?

An identifier is a sequence of letters and digits; the first character must be a letter. The underscore _ counts as a letter. Upper- and lowercase letters are different. If the input stream has been parsed into tokens up to a given character, the next token is taken to include the longest string of characters that could possibly constitute a token. Blanks, tabs, newlines, and comments are ignored except as they serve to separate tokens. Some white space is required to separate otherwise adjacent identifiers, keywords, and constants.

*English description of identifiers in Java.

Regular expressions for some tokens

```
if                                 { IF };
[a-z][a-z0-9]*                     { ID };
[0-9]+                             { NUM };
([0-9]+"."[0-9]*)|([0-9]*"."[0-9]+ { REAL };
("--"[a-z]*"\n")|(" "|"\n"|"\t")+  { scan lexbuf };
 _                                 { error(); scan lexbuf };
```

ocamllex 12-7

Two disambiguation rules

```
if                                 { IF };
[a-z][a-z0-9]*                     { ID };
[0-9]+                             { NUM };
([0-9]+"."[0-9]*)|([0-9]*"."[0-9]+ { REAL };
("--"[a-z]*"\n")|(" "|"\n"|"\t")+  { scan lexbuf };
 _                                 { error(); scan lexbuf };
```

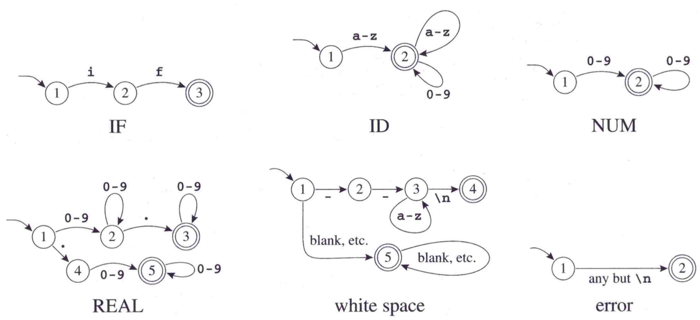
Longest match. The longest initial substring of the input that canmatch any regular expression is taken as the next token.

Rule Priority. For a particular longest initial substring, the first regular expression that can match determines its token.

ocamllex 12-8

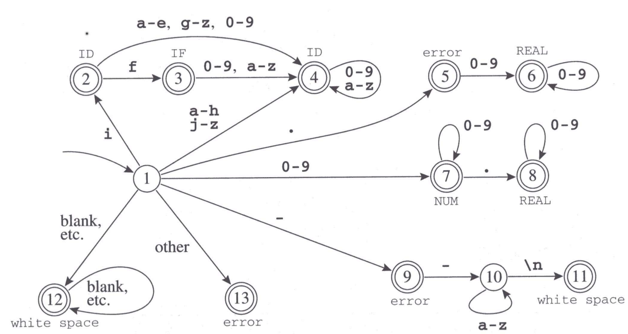
Finite automata

```
if                                        { IF };
[a-z][a-z0-9]*                            { ID };
[0-9]+                                    { NUM };
([0-9]+"."[0-9]*)|([0-9]*"."[0-9]+        { REAL };
("--"[a-z]*"\n")|(" "|"\n"|"\t")+         { scan lexbuf };
_                                         { error(); scan lexbuf };
```

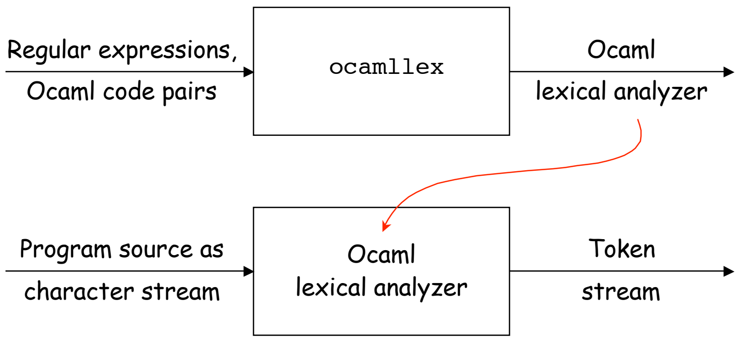


Combining finite automata

```
if                                        { IF };
[a-z][a-z0-9]*                            { ID };
[0-9]+                                    { NUM };
([0-9]+"."[0-9]*)|([0-9]*"."[0-9]+        { REAL };
("--"[a-z]*"\n")|(" "|"\n"|"\t")+         { scan lexbuf };
_                                         { error(); scan lexbuf };
```



ocamllex



A simple example

Header section: OCaml code copied to beginning of output file

```
{
(* An example from ocamllex tutorial *)
  let num_lines = ref 0
  let num_chars = ref 0
}

rule count = parse
| '\n'   { incr num_lines; incr num_chars; count lexbuf }
| _      { incr num_chars; count lexbuf }
|  eof   { () }

{
  let main () =
    let lexbuf = Lexing.from_channel stdin in
    count lexbuf;
    Printf.printf "# of lines = %d, # of chars = %d\n" !num_lines !num_chars

    let _ = Printexc.print main ()
}
```

Trailer section: OCaml code copied to end of output file

The lexer's heart

```
{
(* An example from ocamllex tutorial *)
  let num_lines = ref 0
  let num_chars = ref 0
}

rule count = parse
| '\n'   { incr num_lines; incr num_chars; count lexbuf }
| _      { incr num_chars; count lexbuf }
|  eof   { () }

{
  let main () =
    let lexbuf = Lexing.from_channel stdin in
    count lexbuf;
    Printf.printf "# of lines = %d, # of chars = %d\n" !num_lines !num_chars

    let _ = Printexc.print main ()
}
```

Rules section: contains a series of entrypoints consisting of pattern { action } pairs

An interpreter friendly version of count

```
{ module Counter = struct
(* A version of count that runs in the ocaml repl by typing:
 * main ()
 * followed by lines to count *)
  let num_lines = ref 0
  let num_chars = ref 0
}

rule count = parse
| '\n'   { incr num_lines; incr num_chars; count lexbuf }
| _      { incr num_chars; count lexbuf }
|  eof   { () }

{
  let main () =
    let lexbuf = Lexing.from_channel stdin in
    count lexbuf;
    Printf.printf "# of lines = %d, # of chars = %d\n" !num_lines !num_chars
 end
}
```

Lexing module contains runtime library for lexers generated by `ocamllex`

Patterns

Pattern	Matches
'c'	the character 'c'
_	any character
eof	end-of-file
"foo"	the literal string "foo"
['x' 'y' 'z']	character set
['a' 'b' 'j'-'o' 'z']	character set with ranges
[^ 'A'-'Z']	a negative character set
[^ 'A'-'Z' '\n']	any char EXCEPT uppercase or newline
r*	zero or more r's, r a regular expression
r+	one or more r's, r a regular expression
r?	zero or one r's, r a regular expression

ocamllex 12-15

More patterns

Pattern	Matches
(r)	parenthesis are used to override precedence
ident	the expansion of ident defined earlier in the declaration section
rs	concatenation of regulars exps r and s
r | s	either an r or an s
r#s	difference of two character sets
r as ident	bind the string matched by r to ident

*The regular expression listed are grouped according to precedence, from highest at the top to lowest at the bottom..

ocamllex 12-16

How inputs are matched

```
rule token = parse
  | "ding"              { print_endline "Ding" }
  | ['a'-'z']+ as word { print_endline ("Word: " ^ word) }
```


First match

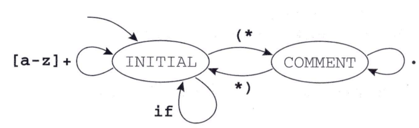
```
rule token = parse
  | ['a'-'z']+ as word { print_endline ("Word: " ^ word) }
  | "ding"              { print_endline "Ding" }
```

Longest match

```
rule token = parse
  | "ding"              { print_endline "Ding" }
  | "dong"              { print_endline "Dong" }
  | "dingdong"          { print_endline "Ding-dong" }
```

ocamllex 12-19

Start conditions




```
{}
rule initial = parse
  | [' ' '\t' '\n']+  { token lexbuf }   (* skip spaces *)
  | "(*"              { comment lexbuf } (* comment rule *)
  . . .
and comment = parse
  | "*)"              { initial lexbuf }(* goto initial *)
  | _                 { comment lexbuf } (* skip comment *)
```

ocamllex 12-20

Keyword hashtable

```
{
  let keyword_table = Hashtbl.create 72
  let _ =
    List.iter (fun (kwd, tok) -> Hastbl.add keyword_table kwd tok)
               [ (“keyword1”, KEYWORD1);
                 (“keyword2”, KEYWORD2);
    …
               ]
}

rule token = parse
  | …
  | [‘A’-’Z’ ‘a’-’z’] [‘A’-’Z’ ‘a’-’z’ ‘0’-’9’ ‘_’]* as id
                        {try
                          Hashtbl.find keyword_table id
                         with
                           Not_found -> IDENT id
}
  | …
```


Nested comments

```
rule token = parse
  | "(*" { print_endline "comments start"; comments 0 lexbuf }
  | [’ ’ ’\t’ ’\n’] { token lexbuf }
  | [’a’-’z’]+ as word  { Printf.printf "word: %s\n" word;
                          token lexbuf }
  | _ as c { Printf.printf "char %c\n" c; token lexbuf }
  | eof { raise End_of_file }

and comments level = parse
  | "*)" { Printf.printf "comments (%d) end\n" level;
           if level = 0 then token lexbuf
           else comments (level-1) lexbuf }
  | "(*" { Printf.printf "comments (%d) start\n" (level+1);
            comments (level+1) lexbuf }
  | _ { comments level lexbuf }
  | eof { print_endline "comments are not closed";
          raise End_of_file }
```

Toy language

```
{
  open Printf
  let create_hashtable size init =
  let tbl = Hashtbl.create size in
  List.iter (fun (key, data) -> Hashtbl.add tbl key data) init;
  type token =
     | IF
     | THEN
     | ELSE
     | BEGIN
     | END
     | FUNCTION
     | ID of string
     | OP of char
     | INT of int
     | FLOAT of float
     | CHAR of char
```


Install our keywords

```
let keyword_table =
  create_hashtable 8 [
     ("if", IF);
     ("then", THEN);
     ("else", ELSE);
     ("begin", BEGIN);
     ("end", END);
     ("function", FUNCTION)
  ]
}

let digit = ['0'-'9']
let id = ['a'-'z' 'A'-'Z']['a'-'z' '0'-'9']*
```

Optional declaration section follows header section

Pattern/action pairs

```
rule toy_lang = parse
  | digit+ as inum { let num = int_of_string inum in
                    printf "integer: %s (%d)\n" inum num;
                    INT num }
  | digit+ '.' digit* as fnum { let num = float_of_string fnum in
                        printf "float: %s (%f)\n" fnum num;
                        FLOAT num }
  | id as word { try
                 let token = Hashtbl.find keyword_table word in
                             printf "keyword: %s\n" word; token
                 with Not_found ->
                       printf "identifier: %s\n" word; ID word }
  | '+'
  | '-'
  | '*'
  | '/' as op   { printf "operator: %c\n" op;  OP op }
```

ocamllex 12-25

Eat comments and whitespace

```
  | '{' [^ '\n']* '}'
  | [' ' '\t' '\n']    { toy_lang lexbuf }
  | _ as c    { printf "Unrecognized character: %c\n" c; CHAR c }
  | eof   { raise End_of_file }
```

ocamllex 12-26

Trailer section

```
{
let rec parse lexbuf =
  let token = toy_lang lexbuf in
  parse lexbuf
```

Get the next token and throw it away; recursively get rest

```
let main () =
  let cin =
    if Array.length Sys.argv > 1
    then open_in Sys.argv.(1)
    else stdin
  in
  let lexbuf = Lexing.from_channel cin in
  try parse lexbuf
  with End_of_file -> ()

let _ = Printexc.print main ()
}
```

Input from file if one is given, else from stdin