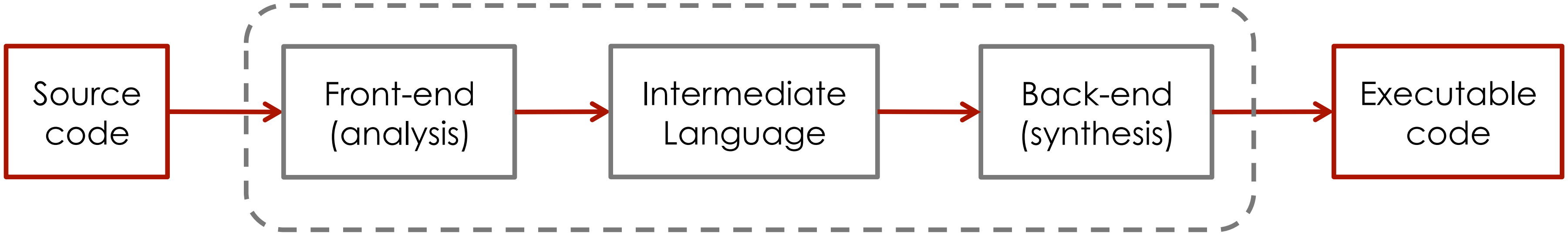
# Lexer and parser generators

Lecture 3

Formal Languages and Compilers 2011

Nataliia Bielova

# Structure of a compiler



Source code

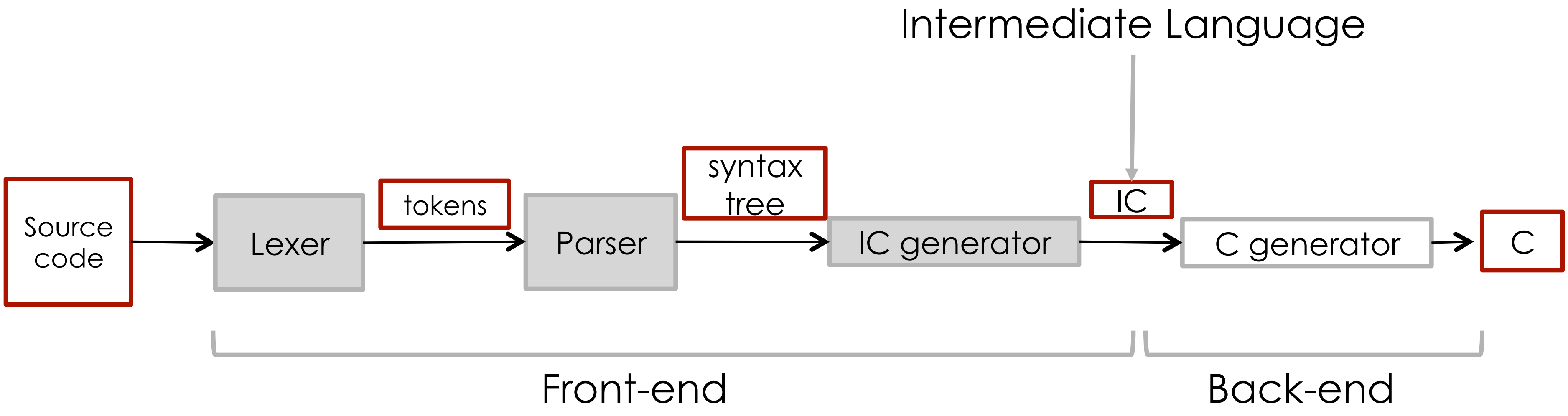
Front-end (analysis)

Intermediate Language

Back-end (synthesis)

Executable code

# Front-end structure



# Lexical analyzer (lexer)

- Input: program in source language
- Output: sequence of tokens (or error)

- Example:

$$17+3*2 \rightarrow \boxed{17}\ \boxed{+}\ \boxed{3}\ \boxed{*}\ \boxed{2}$$

# ocamllex

Generator of lexical analyzer

- Input: “semantic operations” associate with regular expressions
- Output: lexer
- Invocation:

  ocamllex <myfile>.mll

  produces <myfile>.ml with the code of the lexer

# Regular expressions

| | |
|---|---|
| ‘a’ | simple character |
| “string” | string |
| eof | end of file |
| _ | (underscore) any character |
| [ ‘d’ -’ g’ ‘m’ -’ s’ ] | character set |
| [^ ‘a’ -’ c ‘ ‘t’ -’ z’ ] | “negated character set” |
| expr1 # expr2 | difference (of two sets) |
| expr* | zero or more expr |
| expr+ | one or more expr |
| expr? | zero or one expr |
| expr1 \| expr2 | either expr1 or expr2 |
| expr1 expr2 | expr1 followed by expr2 |
| expr as ident | bind the matched string to ident |

# Semantic operations

- Can contain any OCaml code which returns a value AND
- Utility of the library Lexing:

| | |
|---|---|
| Lexing.lexeme lexbuf | string recognized by regexp |
| Lexing.lexeme_char lexbuf n | n-th character of the matched string |
| Lexing.lexeme_start lexbuf | position in which the matched string starts |

…

# Example: calc_lexer.mll

```
{  open Calc_parser (* the type token is in the module calc_parser.mly *)
   exception Eof
}
let white_space = [' ']
rule token = parse
     white_space              { token lexbuf }   (* skip the white space *)
    | ['\n']                  { EOL }
    | ['0'-'9']+ as lxm       { INT(int_of_string lxm ) }
    | '+'                     { PLUS }
    | '*'                     { TIMES }
    | eof                     { raise Eof }
```

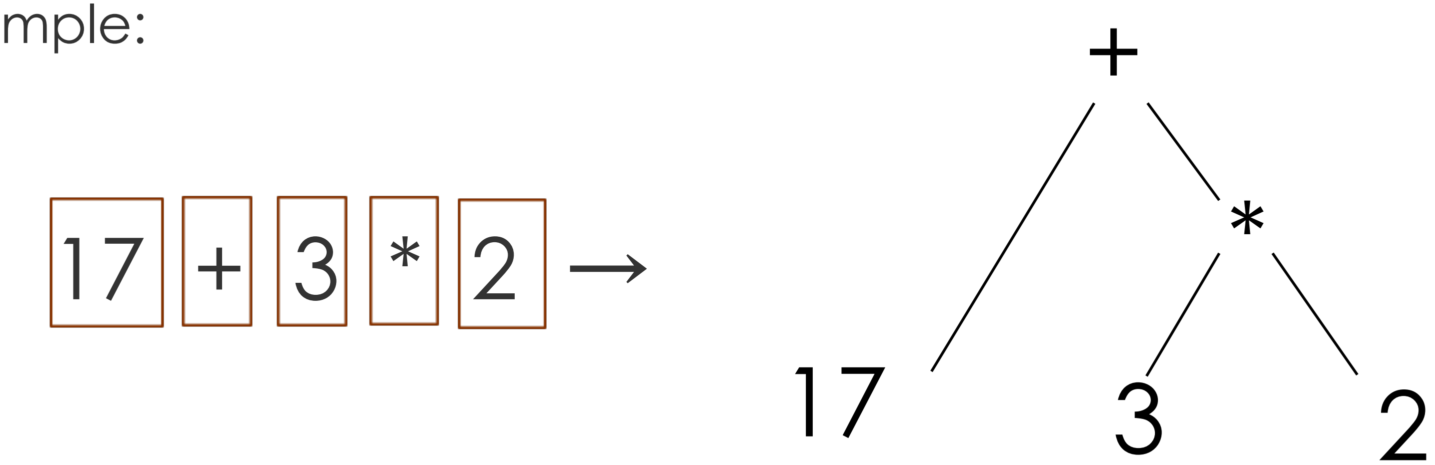
# Structure of the .mll file

```
(* header section *)
  { header }

(* definitions section *)
  let ident = regexp
  let ...

(* rules section *)
  rule entrypoint [arg1... argn] = parse
          | pattern { action }
          | ...
          | pattern { action }
  and entrypoint [arg1... argn] = parse
          ...
  and ...

(* trailer section *)
  { trailer }
```

# Syntactical analyzer (parser)

- Input: sequence of tokens (from lexer)
- Output: parse tree (or syntax tree)

Example:



# ocamlyacc

- Generator of syntactic analyzer (*Yet Another Compiler Compiler*)
- Input: semantic actions associate with context-free grammar
- Output: parser
- Invocation:

  ocamlyacc <myfile>.mly

  produces <myfile>.ml with the code of the parser

# Grammar and semantic actions

- Context-free grammar: puts together terminal and non-terminal symbols

  e.g. expr PLUS expr

- Semantic action: Ocaml code that does the job

# Structure of the .mly file

```
% {
   header (OCaml code)
% }
   declarations  (%token,  %type,  ...)>
%%
   rules  (symbol {semantic action})>
%%
   trailer  (Ocaml code)
```

Comments are enclosed between /* and */ (as in C) in the "declarations" and "rules" sections, and between (* and *) (as in Caml) in the "header" and "trailer" sections.

# Declarations

%token *name… name /* terminal symbols */*

%token < *type*> *name… name /* terminal symbols of*
*specific type*/*

%start *symbol … symbol /* nonterminal starting symbol,, for*
*which type should be defined*/*

%type < *type*> *symbol … symbol /* declare type of*
*nonterminal symbol */*

%left *symbol … symbol*

%right *symbol … symbol*

%nonassoc *symbol … symbol*

# Rules

```
nonterminal :
    symbol … symbol { semantic-action }
    | …
    | symbol … symbol { semantic-action }
;
```

Semantic actions

- are arbitrary Caml expressions
- can access the semantic attributes with the $ notation:
  expr PLUS expr { $1 + $3 }

# Example: calc_parser.mly

```
%token <int> INT
%token PLUS TIMES
%token EOL
%left PLUS   /* lower precedence */
%left TIMES  /* higher precedence */

%start main
%type <int> main

%%
main:
    expr EOL                    { $1 }
;
expr:
     INT                        { $1 }
    | expr PLUS expr            { $1 + $3 }
    | expr TIMES expr           { $1 * $3 } ;
;
```

# Calculator

http://disi.unitn.it/~bielova/flc/exercises/03-Calculator.zip

- Definition of the lexer: calc_lexer.mll
- Definition of the parser: calc_parser.mly
- Main program: calc_main.ml

Compilation:

```
ocamllex  calc_lexer.mll  #  generates  calc_lexer.ml
ocamlyacc  calc_parser.mly  #  generates  calc_parser.ml  and  calc_parser.mli
ocamlc  -c  calc_parser.mli
ocamlc  -c  calc_lexer.ml
ocamlc  -c  calc_parser.ml
ocamlc  -c  calc_main.ml
ocamlc  -o  calc calc_lexer.cmo  calc_parser.cmo  calc_main.cmo
./calc
```

# Excercise

Extend the calculator with:

- Add tabulations to the white spaces
- Add subtraction and division
- Add unary function “-”
- Parenthesis
- Change the syntax to prefix syntax:

  + * 3 4 5 = 17
- Add an operator with arbitrary number of operands:

  (+ (* 1 2 3) 4 5 ) = 15
- Try whatever you like