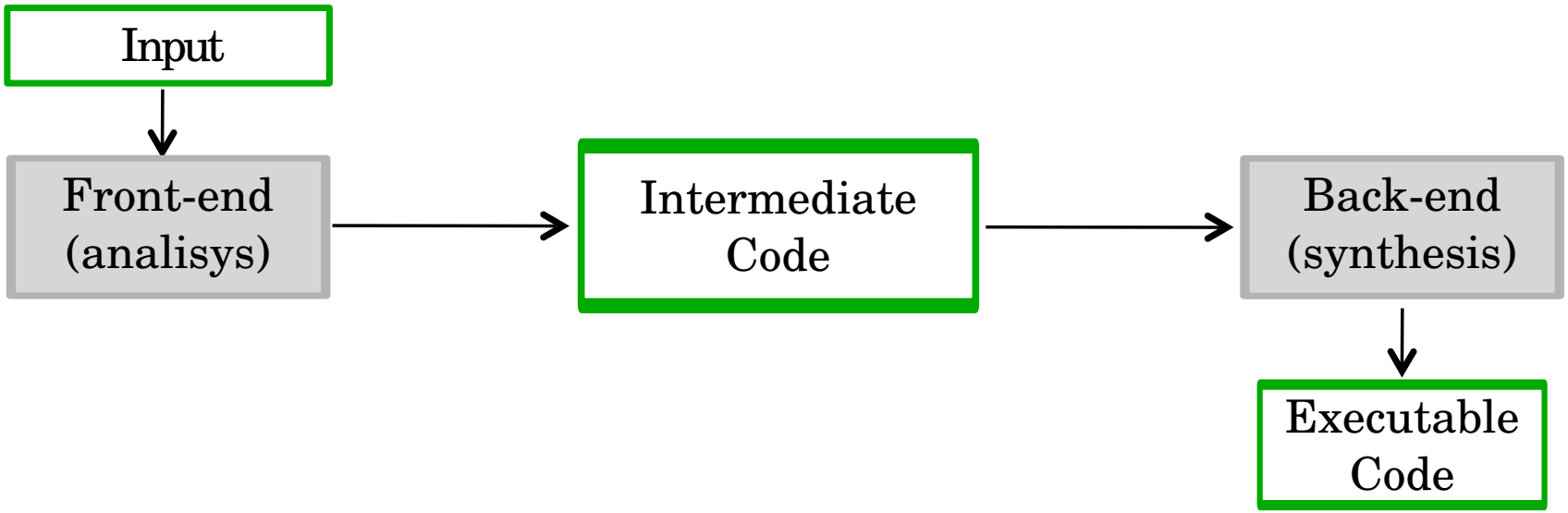
Formal Languages and Compilers - Exercises
Lecture 3
Lexer and parser generators

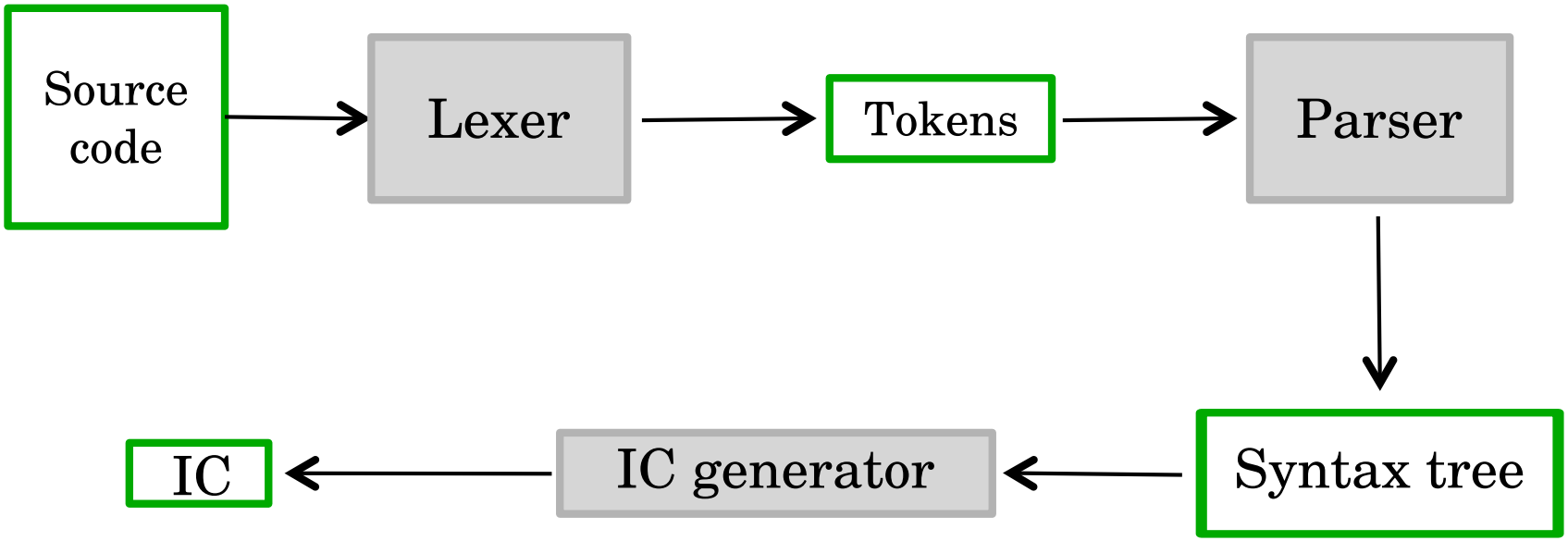
27/03/2012

Outline

Structure of a compiler



Front-end details



Outline

Lexical analyzer (lexer)

Input: program in source language

Output: sequence of tokens (or error)

Example

From the expression $17 + 3 * 2$
obtain the set of tokens $(17, +, 3, *, 2)$

Lexical analyzer (lexer)

Input: program in source language

Output: sequence of tokens (or error)

Example

From the expression $17 + 3 * 2$
obtain the set of tokens (17, +, 3, *, 2)

ocamllex

Generator of lexical analyzer

Input: semantic operations associate with regular expressions

Output: lexer

Invocation: `ocamllex <myfile>.mll` produces `<myfile>.ml` which contains 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
- Utility of the Lexing library

Some Lexing functions

- 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

Semantic operations

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

Some `Lexing` functions

- 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

Structure of the .mll file

```
(* header section *)
    { header }
(* definitions section *)
    let ident = regexp
    let ...
(* rules section *)
    rule entrypoint [arg1... argn] = parse
        pattern1 { action1 }
      | ...
      | pattern2 { action2 }
    and entrypoint [arg1... argn] = parse
      ...
(* trailer section *)
    { trailer }
```

Example: calc_lexer.mll

```
{
  open Calc_parser
  exception Eof
}

  let white_space = [' ']
  rule token = parse
      white_space          { token lexbuf }
    | ['\n']               { EOL }
    | ['0'-'9']+ as lxm    { INT(int_of_string lxm) }
    | '+'                  { PLUS }
    | '*'                  { TIMES }
    | eof                  { raise Eof }
```

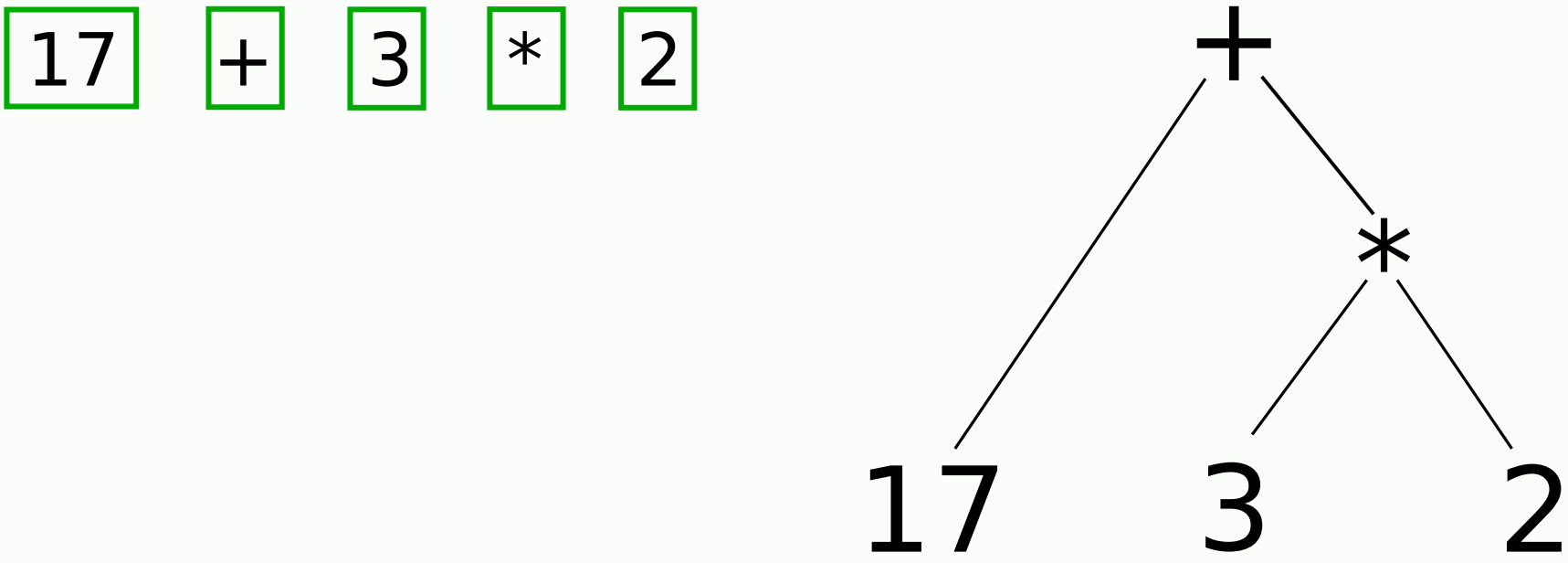
Outline

Syntactical analyzer (parser)

Input: sequence of tokens (or error)

Output: syntax tree (or parse tree)

Example



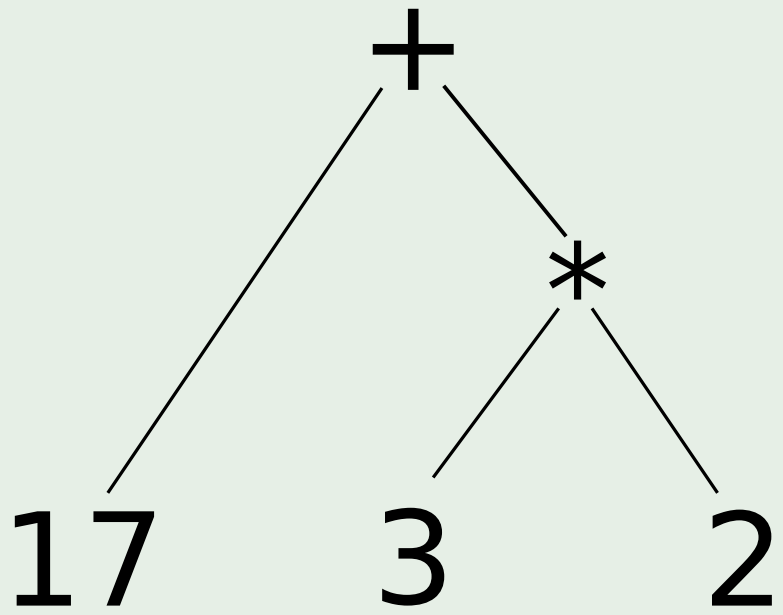
Syntactical analyzer (parser)

Input: sequence of tokens (or error)

Output: syntax tree (or parse tree)

Example

17 + 3 * 2



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: O'CaML code that does the job

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: O'CaML 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 O'CaML) in the header and trailer sections.

Declarations - 1

Terminal symbols (with optional type)

```
%token              name, ..., name
%token <type>       name, ..., name
```

Non-terminal starting symbol (have to define type for it)

```
%start  symbol ... symbol
```

Declarations - 1

Terminal symbols (with optional type)

```
%token              name, ..., name
%token <type>       name, ..., name
```

Non-terminal starting symbol (have to define type for it)

```
%start symbol ... symbol
```

Declarations - 2

Type definition for non-terminal symbol

```
%type <type>  symbol ... symbol
```

Associativity of symbols

```
%left symbol ... symbol
%right symbol ... symbol
%nonassoc symbol ... symbol
```

Declarations - 2

Type definition for non-terminal symbol

```
%type <type>  symbol ... symbol
```

Associativity of symbols

```
%left symbol ... symbol
%right symbol ... symbol
%nonassoc symbol ... symbol
```

Rules

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

Semantic actions

- Are arbitrary O'CaML expressions
- Can access the semantic attributes with the $ notation:
  `expr PLUS expr { $1 + $3 }`

Example: calc_parser.mly - Header

```
/* parser */
%token <int> INT
%token PLUS TIMES
%token EOL

%left PLUS    /* lower precedence */
%left TIMES   /* higher precedence */

%start main
%type <int> main

%%
 ...
```

Example: calc_parser.mly - Header

```
...
main:
      expr EOL                              { $1 }
;

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

Outline

Hands on: calculator

Structures

calc_lexer.mll: Definition of the lexer

calc_parser.mly: Definition of the parser

main.ml: Main program

Hands on: calculator

Compilation

```
ocamllex    calc_lexer.mll
ocamlyacc   calc_parser.mly
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
```

Or, simply, `make all`

Hands on: calculator

Compilation

```
ocamllex    calc_lexer.mll
ocamlyacc   calc_parser.mly
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
```

Or, simply, `make all`

Exercise time!

Extend the calculator with

- Add tabulations to the white spaces
- Add subtraction and division
- Add unary function -
- Parenthesis
- Change the syntax to prefix syntax:
  $+ * 345 = 17$
- Add an operator with arbitrary number of operands:
  $(+(*123)45) = 15$
- Whatever you want, whatever you like...