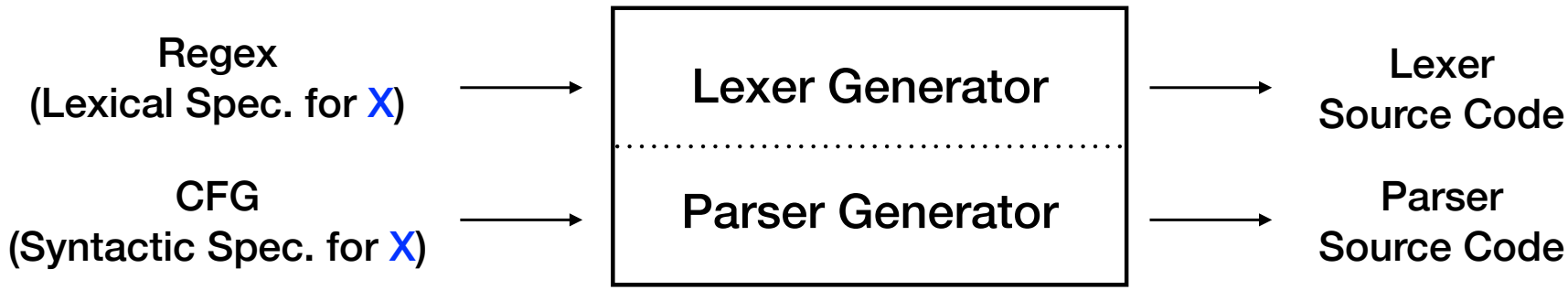
EC3204: Programming Languages and Compilers

Lecture 9 — Lexer & Parser Generators

Sunbeom So
Fall 2024

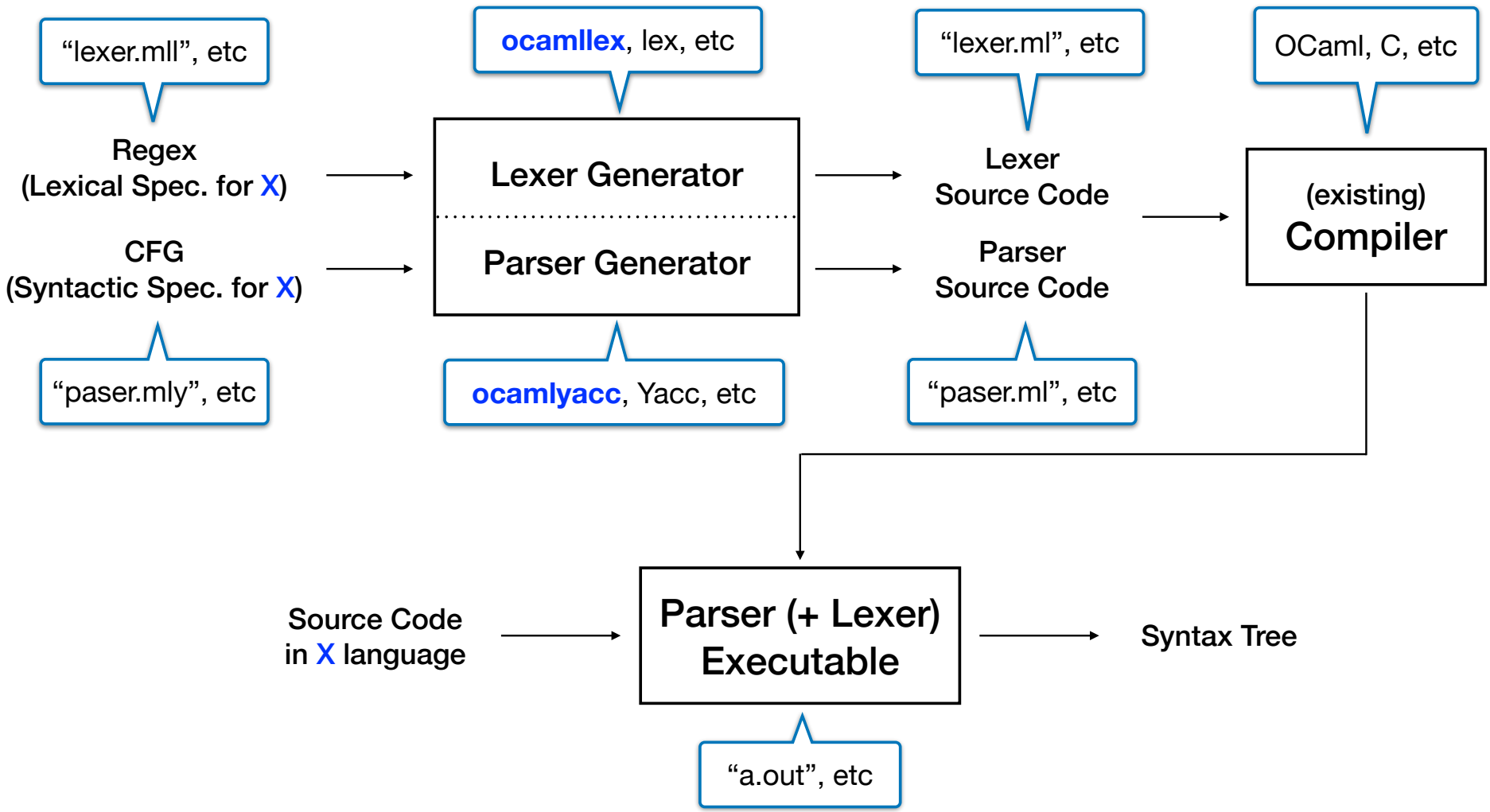
Goal: Useful Tools for Compiler Construction

- We have learned lexing and parsing algorithms.
  - Lexer: Thompson's construction and subset construction
  - Parser: top-down and bottom-up parsing algorithms
- Fortunately, there are useful tools for compiler construction; you do not need to implement those lexing/parsing algorithms by hand.
- We will learn how to use `ocamllex` and `ocamlyacc`.
  - `ocamlyacc`: a (LALR) parser generator for OCaml[1]
  - `ocamllex`: a lexer generator for OCaml



[1] for lexing/parsing a new language "X" using OCaml (O), for building an OCaml Compiler (X)

Compiler Construction with ocamllex and ocamlyacc



Example: Calculator

We focus on implementing a calculator that parses arithmetic expressions.

```
1 $ ./a.out
2 1+2*3
3 7
```

Our target source language (arithmetic expression) is defined as follows:

$$\boldsymbol{E} \rightarrow \boldsymbol{E} + \boldsymbol{E} \mid \boldsymbol{E} * \boldsymbol{E} \mid (\boldsymbol{E}) \mid \textbf{number}$$

The implementation consists of four files:

- `ast.ml`: definitions of abstract syntax tree and evaluator
- `parser.mly`: the input (context-free grammar) to `ocamlyacc`
- `lexer.mll`: the input (regular expressions) to `ocamllex`
- `main.ml`: the driver routine

ast.ml

```
 1 type expr =
 2   | Num of int
 3   | Add of expr * expr
 4   | Mul of expr * expr
 5
 6 let rec eval : expr -> int
 7 = fun e ->
 8    match e with
 9    | Num n -> n
10    | Add (e1,e2) -> (eval e1) + (eval e2)
11    | Mul (e1,e2) -> (eval e1) * (eval e2)
```

Layout of `parser.mly`

```
1 %{
2   User declarations
3 %}
4   Parser declarations
5 %%
6   Grammar rules
```

- User declarations: OCaml declarations usable from the parser
- Parser declarations: terminal and nonterminal symbols, precedence, associativity, etc.
- Grammar rules: productions of the grammar.

parser.mly

```
1  %{
2  %}
3
4  %token NEWLINE LPAREN RPAREN PLUS MINUS MULTIPLY /*token names (terminals)*/
5  %token <int> NUM /* when the token has values, its type must be annotated */
6
7
8  %start program /* start symbol (entry point) of the grammar */
9  %type <Ast.expr> program /* type annotation is mandatory for start symbols */
10
11 %%
12
13 program : exp NEWLINE { $1 } /* productions of the grammar (L13-19) */
14
15 exp:
16   | NUM { Ast.Num ($1) } /* {...} indicates the``action'' part */
17   | exp PLUS exp  { Ast.Add ($1,$3) } /* do translation (whenever reduce) */
18   | exp MULTIPLY exp  { Ast.Mul ($1,$3) }
19   | LPAREN exp RPAREN { $2 } /* $2: the attribute for the second symbol */
```

- Lines 13–19: $S \rightarrow e,\ E \rightarrow \mathbf{number} \mid E + E \mid E * E \mid (E)$
- Q. What are the meaning of the tokens (e.g., PLUS, LPAREN)?

lexer.mll

```
1  {
2    open Parser
3    exception LexicalError
4  }
5
6  /* regular definitions */
7  let number = ['0'-'9']+
8  let blank = [' ' '\t']
9
10 rule token = parse
11   | blank  { token lexbuf } /* recursion on the remaining input stream */
12   | '\n'   { NEWLINE }
13   /* lexeme: returns the string matched by a given regular expression */
14   | number { NUM (int_of_string (Lexing.lexeme lexbuf)) }
15   | '+'    { PLUS }
16   | '-'    { MINUS }
17   | '*'    { MULTIPLY }
18   | '('    { LPAREN }
19   | ')'    { RPAREN }
20   | _      { raise LexicalError }
```

main.ml

```
1 let main() =
2   let lexbuf = Lexing.from_channel stdin in
3   let ast = Parser.program Lexer.token lexbuf in
4   let num = Ast.eval ast in
5   print_endline (string_of_int num)
6
7 let _ = main ()
```

- `Lexing.from_channel stdin` (line 2): returns a buffer from a standard input[2]
- `Lexer.token` (line 3): automatically generated lexer that converts input characters into tokens
- `Parser.program` (line 3): automatically generated parser that converts a token sequence into a syntax tree

cf) `Lexer.token` and `Parser.program` are not fixed names.

[2] https://ocaml.org/manual/5.1/api/Lexing.html

Build Script (Makefile)

All commands necessary for producing a lexer `lexer.ml` and a parser `parser.ml` are merged into `Makefile`.

```
1  all:
2    ocamlc -c ast.ml
3    ocamlyacc parser.mly
4    ocamlc -c parser.mli
5    ocamllex lexer.mll
6    ocamlc -c lexer.ml
7    ocamlc -c parser.ml
8    ocamlc -c main.ml
9    ocamlc ast.cmo lexer.cmo parser.cmo main.cmo
10
11   clean:
12     rm -f *.cmo *.cmi a.out lexer.ml parser.ml parser.mli
```

Conflicts

```
$ make
ocamlc -c ast.ml
ocamlyacc parser.mly
4 shift/reduce conflicts.
ocamlc -c parser.mli
ocamllex lexer.mll
10 states, 267 transitions, table size 1128 bytes
ocamlc -c lexer.ml
ocamlc -c parser.ml
ocamlc -c main.ml
ocamlc ast.cmo lexer.cmo parser.cmo main.cmo
```

- Q. Why are there shift/reduce conflicts?

Conflicts

```
1  $ make
2  ocamlc -c ast.ml
3  ocamlyacc parser.mly
4  4 shift/reduce conflicts.
5  ocamlc -c parser.mli
6  ocamllex lexer.mll
7  10 states, 267 transitions, table size 1128 bytes
8  ocamlc -c lexer.ml
9  ocamlc -c parser.ml
10 ocamlc -c main.ml
11 ocamlc ast.cmo lexer.cmo parser.cmo main.cmo
```

- Q. Why are there shift/reduce conflicts?
- A. We have not defined the precedence between $+$ and $*$.

Conflicts

```
$ make
ocamlc -c ast.ml
ocamlyacc parser.mly
4 shift/reduce conflicts.
ocamlc -c parser.mli
ocamllex lexer.mll
10 states, 267 transitions, table size 1128 bytes
ocamlc -c lexer.ml
ocamlc -c parser.ml
ocamlc -c main.ml
ocamlc ast.cmo lexer.cmo parser.cmo main.cmo
```

- Q. Why are there shift/reduce conflicts?
- A. We have not defined the precedence between $+$ and $*$.
- Q. What happens due to these conflicts? Demonstrate the problems with concrete examples.

Conflicts

```
$ make
ocamlc -c ast.ml
ocamlyacc parser.mly
4 shift/reduce conflicts.
ocamlc -c parser.mli
ocamllex lexer.mll
10 states, 267 transitions, table size 1128 bytes
ocamlc -c lexer.ml
ocamlc -c parser.ml
ocamlc -c main.ml
ocamlc ast.cmo lexer.cmo parser.cmo main.cmo
```

- Q. Why are there shift/reduce conflicts?
- A. We have not defined the precedence between $+$ and $*$.
- Q. What happens due to these conflicts? Demonstrate the problems with concrete examples.
- A. 2*3+1 evaluates to 8.

Conflicts

```
1  $ make
2  ocamlc -c ast.ml
3  ocamlyacc parser.mly
4  4 shift/reduce conflicts.
5  ocamlc -c parser.mli
6  ocamllex lexer.mll
7  10 states, 267 transitions, table size 1128 bytes
8  ocamlc -c lexer.ml
9  ocamlc -c parser.ml
10 ocamlc -c main.ml
11 ocamlc ast.cmo lexer.cmo parser.cmo main.cmo
```

- Q. Why are there shift/reduce conflicts?
- A. We have not defined the precedence between $+$ and $*$.
- Q. What happens due to these conflicts? Demonstrate the problems with concrete examples.
- A. 2*3+1 evaluates to 8.
- Q. Why 8?

Conflicts

```
1  $ make
2  ocamlc -c ast.ml
3  ocamlyacc parser.mly
4  4 shift/reduce conflicts.
5  ocamlc -c parser.mli
6  ocamllex lexer.mll
7  10 states, 267 transitions, table size 1128 bytes
8  ocamlc -c lexer.ml
9  ocamlc -c parser.ml
10 ocamlc -c main.ml
11 ocamlc ast.cmo lexer.cmo parser.cmo main.cmo
```

- Q. Why are there shift/reduce conflicts?
- A. We have not defined the precedence between $+$ and $*$.
- Q. What happens due to these conflicts? Demonstrate the problems with concrete examples.
- A. 2*3+1 evaluates to 8.
- Q. Why 8?
- A. General rule in (most) parser generators: prefer shift over reduce.

Conflicts

```
1  $ make
2  ocamlc -c ast.ml
3  ocamlyacc parser.mly
4  4 shift/reduce conflicts.
5  ocamlc -c parser.mli
6  ocamllex lexer.mll
7  10 states, 267 transitions, table size 1128 bytes
8  ocamlc -c lexer.ml
9  ocamlc -c parser.ml
10 ocamlc -c main.ml
11 ocamlc ast.cmo lexer.cmo parser.cmo main.cmo
```

- Q. Why are there shift/reduce conflicts?
- A. We have not defined the precedence between $+$ and $*$.
- Q. What happens due to these conflicts? Demonstrate the problems with concrete examples.
- A. 2*3+1 evaluates to 8.
- Q. Why 8?
- A. General rule in (most) parser generators: prefer shift over reduce.
- Q. How can we resolve the conflicts?

Fixed parser.mly

```
1  %{
2  %}
3
4  %token NEWLINE LPAREN RPAREN PLUS MINUS MULTIPLY /*token names (terminals)*/
5  %token <int> NUM /* when the token has values, its type must be annotated */
6
7  %left PLUS /* lower precedence */
8  %left MULTIPLY /* higher precedence */
9
10 %start program /* start symbol (entry point) of the grammar */
11 %type <Ast.expr> program /* type annotation is mandatory for start symbols */
12
13 %%
14
15 program : exp NEWLINE { $1 } /* productions of the grammar (L13-19) */
16
17 exp:
18   | NUM { Ast.Num ($1) } /* {...} indicates the``action'' part */
19   | exp PLUS exp  { Ast.Add ($1,$3) } /* do translation (whenever reduce) */
20   | exp MULTIPLY exp  { Ast.Mul ($1,$3) }
21   | LPAREN exp RPAREN { $2 } /* $2: the attribute for the second symbol */
```

Execution Results After the Fix

```
$ make
ocamlc -c ast.ml
ocamlyacc parser.mly
ocamlc -c parser.mli
ocamllex lexer.mll
10 states, 267 transitions, table size 1128 bytes
ocamlc -c lexer.ml
ocamlc -c parser.ml
ocamlc -c main.ml
ocamlc ast.cmo lexer.cmo parser.cmo main.cmo
$ ./a.out
2*3+1
7
$ ./a.out
5+2*100+1
206
$ ./a.out
1+(2+3)*5
26
```

Summary & Exercises

- Useful tools for helping compiler construction: lexer/parser generators.
- (Exercise 1) Modify `parser.mly` to give precedence to $+$ over $*$.
- (Exercise 2) Extend `lexer.mll` and `parser.mly` to parse boolean expressions:

$$\begin{aligned} \boldsymbol{B} &\rightarrow \boldsymbol{E} > \boldsymbol{E} \mid \boldsymbol{E} = \boldsymbol{E} \mid \mathbf{true} \mid \mathbf{false} \\ \boldsymbol{E} &\rightarrow \boldsymbol{E} + \boldsymbol{E} \mid \boldsymbol{E} * \boldsymbol{E} \mid (\boldsymbol{E}) \mid \mathbf{number} \end{aligned}$$

  where the start variable is $\boldsymbol{B}$.