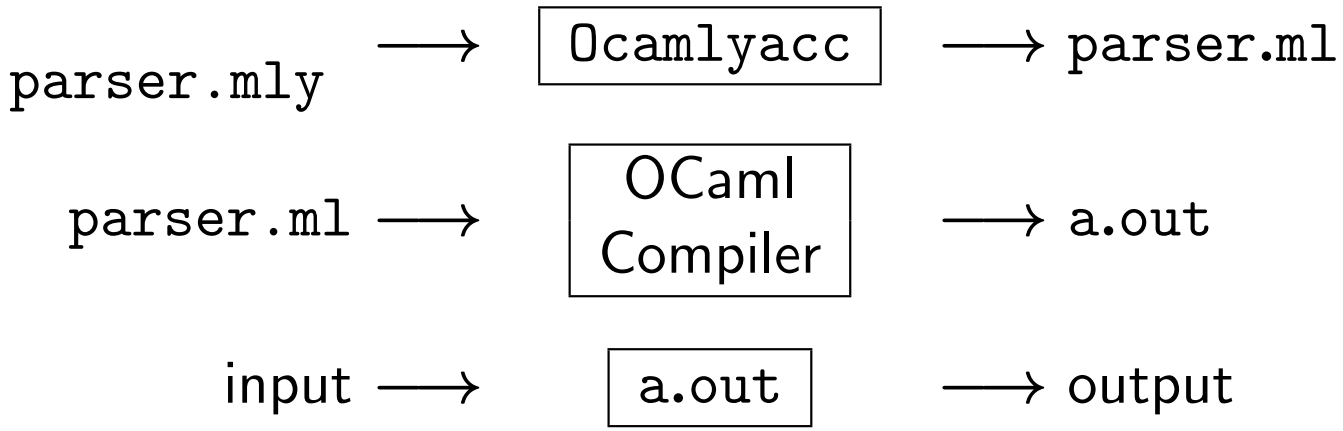
COSE312: Compilers

Lecture 10 — Using Parser Generators

Hakjoo Oh
2017 Spring

Yacc: "Yet Another Compiler-Compiler"

- `Yacc`: a parser generator for C
- `Ocamlyacc`: a parser generator for OCaml



Example: Calculator

The source language:

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

An example execution:

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

The implementation consists of four files:

- `ast.ml`: definitions of abstract syntax tree and evaluator
- `parser.mly`: the input to Ocamlyacc
- `lexer.mll`: the input 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)
```

Grammar Specification

```
%{
  User declarations
%}
  Parser declarations
%%
  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

```
%{
%}

%token NEWLINE LPAREN RPAREN PLUS MINUS MULTIPLY
%token <int> NUM

%start program
%type <Ast.expr> program

%%

program : exp NEWLINE { $1 }

exp: NUM { Ast.Num ($1) }
| exp PLUS exp { Ast.Add ($1,$3) }
| exp MULTIPLY exp  { Ast.Mul ($1,$3) }
| LPAREN exp RPAREN { $2 }
```

lexer.mll

```
 1 {
 2   open Parser
 3   exception LexicalError
 4 }
 5
 6 let number = ['0'-'9']+
 7 let blank = [' ' '\t']
 8
 9 rule token = parse
10   | blank  { token lexbuf }
11   | '\n'   { NEWLINE }
12   | number { NUM (int_of_string (Lexing.lexeme lexbuf)) }
13   | '+'    { PLUS }
14   | '-'    { MINUS }
15   | '*'    { MULTIPLY }
16   | '('    { LPAREN }
17   | ')'    { RPAREN }
18   | _      { 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 ()
```

Build

```
 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
```

parser.mly

```
 1 %{
 2 %}
 3
 4 %token NEWLINE LPAREN RPAREN PLUS MINUS MULTIPLY
 5 %token <int> NUM
 6
 7 %left PLUS
 8 %left MULTIPLY
 9
10 %start program
11 %type <Ast.expr> program
12
13 %%
14
15 program : exp NEWLINE { $1 }
16
17 exp:  NUM   { Ast.Num ($1) }
18   | exp PLUS exp  { Ast.Add ($1,$3) }
19   | exp MULTIPLY exp  { Ast.Mul ($1,$3) }
20   | LPAREN exp RPAREN { $2 }
```

Execution

```
$ 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
  calc  ./a.out
1+(2+3)*5
26
```