COSE312: Compilers

Lecture 7 — Using Parser Generators

Hakjoo Oh
2025 Spring

Contents

- Writing a simple calculator
- Writing a simple programming language

Yacc: "Yet Another Compiler-Compiler"

- `yacc`: a parser generator for C
- `ocamlyacc`: a parser generator for OCaml

`parser.mly`	$\longrightarrow$	$\boxed{\texttt{ocamlyacc}}$	$\longrightarrow$ `parser.ml`
`parser.ml`	$\longrightarrow$	$\boxed{\texttt{ocamlc}}$	$\longrightarrow$ `a.out`
input	$\longrightarrow$	$\boxed{\texttt{a.out}}$	$\longrightarrow$ output

Example: Calculator

Abstract syntax:

$$E \rightarrow n \mid E + E \mid E - E \mid E * E \mid E/E \mid E^{E}$$

Example runs:

```
$ ./a.out
1 + 2 * 3
7
$ ./a.out
1 - 2 - 3
-4
$ ./a.out
(1 + 2) ^ 3 * 4
108
$ ./a.out
(2 ^ 3) ^ 4
4096
```

Example: Calculator

The implementation consists of the files:

- `ast.ml`: abstract syntax
- `eval.ml`: evaluator implementation
- `parser.mly`: the input to ocamlyacc
- `lexer.mll`: the input to ocamllex
- `main.ml`: the driver routine

ast.ml

```
type expr =
   Num of int
 | Add of expr * expr
 | Sub of expr * expr
 | Mul of expr * expr
 | Div of expr * expr
 | Pow of expr * expr
```

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 DIV POW
%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 MINUS exp { Ast.Sub ($1, $3) }
| exp MULTIPLY exp { Ast.Mul ($1, $3) }
| exp DIV exp { Ast.Div ($1, $3) }
| exp POW exp { Ast.Pow ($1, $3) }
| LPAREN exp RPAREN { $2 }
```

lexer.mll

```
{
  open Parser
  exception LexicalError
}

let number = ['0'-'9']+
let blank = [' ' '\t']

rule token = parse
  | blank { token lexbuf }
  | '\n' { NEWLINE }
  | number { NUM (int_of_string (Lexing.lexeme lexbuf)) }
  | '+' { PLUS }
  | '-' { MINUS }
  | '*' { MULTIPLY }
  | '/' { DIV }
  | '^' { POW }
  | '(' { LPAREN }
  | ')' { RPAREN }
  | _ { raise LexicalError }
```

eval.ml

```
open Ast

let rec eval : expr -> int
=fun e ->
  match e with
  | Num n -> n
  | Add (e1, e2) -> (eval e1) + (eval e2)
  | Sub (e1, e2) -> (eval e1) - (eval e2)
  | Mul (e1, e2) -> (eval e1) * (eval e2)
  | Div (e1, e2) -> (eval e1) / (eval e2)
  | Pow (e1, e2) -> pow (eval e1) (eval e2)

and pow a b =
  if b = 0 then 1 else a * pow a (b-1)
```

main.ml

```
let main () =
  let lexbuf = Lexing.from_channel stdin in
  let ast = Parser.program Lexer.token lexbuf in
  let num = Eval.eval ast in
    print_endline (string_of_int num)

let _ =  main ()
```

Makefile

```
all:
  ocamlc -c ast.ml
  ocamlyacc parser.mly
  ocamlc -c parser.mli
  ocamllex lexer.mll
  ocamlc -c lexer.ml
  ocamlc -c parser.ml
  ocamlc -c eval.ml
  ocamlc -c main.ml
  ocamlc ast.cmo lexer.cmo parser.cmo eval.cmo main.cmo

clean:
  rm -f *.cmo *.cmi a.out lexer.ml parser.ml parser.mli
```

Conflicts

```
$ make
ocamlc -c ast.ml
ocamlyacc parser.mly
25 shift/reduce conflicts.
ocamlc -c parser.mli
ocamllex lexer.mll
12 states, 267 transitions, table size 1140 bytes
...

$ ./a.out
1 + 2 * 3
7
$ ./a.out
1 - 2 - 3
2
$ ./a.out
(1 + 2) ^ 3 * 4
531441
$ ./a.out
(2 ^ 3) ^ 4
4096
```

Resolving Conflicts

```
%{
%}

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

%left PLUS MINUS
%left MULTIPLY DIV
%right POW

%start program
%type <Ast.expr> program

%%

program : exp NEWLINE { $1 }

exp : NUM { Ast.Num ($1) }
| exp PLUS exp { Ast.Add ($1, $3) }
| exp MINUS exp { Ast.Sub ($1, $3) }
| exp MULTIPLY exp { Ast.Mul ($1, $3) }
| exp DIV exp { Ast.Div ($1, $3) }
| exp POW exp { Ast.Pow ($1, $3) }
| LPAREN exp RPAREN { $2 }
```

Resolving Conflicts

```
$ make
ocamlc -c ast.ml
ocamlyacc parser.mly
ocamlc -c parser.mli
ocamllex lexer.mll
10 states, 267 transitions, table size 1128 bytes
...

$ ./a.out
1 + 2 * 3
7
$ ./a.out
1 - 2 - 3
-4
$ ./a.out
(1 + 2) ^ 3 * 4
108
$ ./a.out
(2 ^ 3) ^ 4
4096
```

Example: The While Language

Abstract syntax:

$$
\begin{array}{rcl}
a & \rightarrow & n \mid x \mid a_1 + a_2 \mid a_1 \star a_2 \mid a_1 - a_2 \mid a1/a2 \mid a1\%a2 \\
b & \rightarrow & \texttt{true} \mid \texttt{false} \mid a_1 = a_2 \mid a_1 \leq a_2 \mid \neg b \mid b_1 \wedge b_2 \\
c & \rightarrow & x := a \mid \texttt{skip} \mid c_1; c_2 \mid \texttt{if}\ b\ c_1\ c_2 \mid \texttt{while}\ b\ c
\end{array}
$$

Examples

```
// sum.c
n := 10; i := 1;
fact := 1;
while (i <= n) {
    fact := fact * i;
    i := i + 1;
}
print (fact);

// fact.c
n := 10; i := 1;
evens := 0; // sum of even numbers
odds := 0;  // sum of odd numbers
while (i <= n)
{
    if (!(i % 2 == 1) && i % 2 == 0) {
        evens := evens + i;
    } else {
        odds := odds + i;
    }
    i := i + 1;
}
print (evens);
print (odds);
```

ast.ml

```
type var = string

type aexp =
| Int of int | Var of var
| Add of aexp * aexp | Sub of aexp * aexp
| Mul of aexp * aexp | Div of aexp * aexp  | Mod of aexp * aexp

type bexp =
| Bool of bool
| Eq of aexp * aexp
| Le of aexp * aexp
| Neg of bexp
| Conj of bexp * bexp

type cmd =
| Assign of var * aexp
| Skip
| Seq of cmd * cmd
| If of bexp * cmd * cmd
| While of bexp * cmd
| Print of aexp

type program = cmd
```

parser.mly

```
%{
%}

%token <string> IDENT
%token <int> NUMBER
%token <bool> BOOLEAN
%token LPAREN RPAREN LBRACE RBRACE SEMICOLON EOF
%token BAND NOT LE EQ PLUS MINUS STAR SLASH MOD ASSIGN SKIP PRINT IF ELSE WHILE

%left   BAND
%left   EQ
%left   LE
%left   PLUS MINUS
%left   STAR SLASH MOD
%right  NOT

%type <Ast.cmd> cmd
%type <Ast.aexp> aexp
%type <Ast.bexp> bexp
%type <Ast.program> program

%start program

%%
```

Exercise

Complete the parser specification by writing production rules.

```
program :
```

```
cmd :
```

```
aexp :
```

```
bexp :
```

lexer.mll

```
{
  open Parser
  exception LexingError of string

  let kwd_list : (string * Parser.token) list =
    [
      ("true", BOOLEAN true);
      ("false", BOOLEAN false);
      ("if", IF);
      ("else", ELSE);
      ("while", WHILE);
      ("skip", SKIP);
      ("print", PRINT)
    ]

  let id_or_kwd (s : string) : Parser.token =
    match List.assoc_opt s kwd_list with
    | Some t -> t
    | None -> IDENT s
}

let letter    = ['a'-'z' 'A'-'Z']
let digit     = ['0'-'9']
let number    = digit+
```

```
let number    = digit+
let space     = ' ' | '\t' | '\r'
let blank     = space+
let new_line  = '\n' | "\r\n"
let ident     = letter (letter | digit | '_')*

let comment_line_header   = "//"

rule next_token = parse
  | comment_line_header   { comment_line lexbuf }
  | blank                 { next_token lexbuf }
  | new_line              { Lexing.new_line lexbuf; next_token lexbuf }
  | ident as s            { id_or_kwd s }
  | number as n           { NUMBER (int_of_string n) }
  | '('                   { LPAREN }
  | ')'                   { RPAREN }
  | '{'                   { LBRACE }
  | '}'                   { RBRACE }
  | ';'                   { SEMICOLON }
  | "=="                  { EQ }
  | "<="                  { LE }
  | '!'                   { NOT }
  | '+'                   { PLUS }
  | '-'                   { MINUS }
  | '*'                   { STAR }
  | '/'                   { SLASH }
```

```
  | '%'                  { MOD }
  | "&&"                 { BAND }
  | ":="                 { ASSIGN }
  | eof                  { EOF }
  | _ as c
    { LexingError (": illegal character \'" ^ (c |> String.make 1) ^ "\'")
      |> Stdlib.raise }

and comment_line = parse
  | new_line             { Lexing.new_line lexbuf; next_token lexbuf }
  | eof                  { EOF }
  | _                    { comment_line lexbuf }
```

eval.ml

```
open Ast

module State = struct
  type t = (var * int) list
  let empty = []
  let rec lookup s x =
  match s with
  | [] -> raise (Failure (x ^ " is not bound in state"))
  | (y,v)::s' -> if x = y then v else lookup s' x
  let update s x v = (x,v)::s
end

let rec eval_a : aexp -> State.t -> int
=fun a s ->
  match a with
  | Int n -> n
  | Var x -> State.lookup s x
  | Add (a1, a2) -> (eval_a a1 s) + (eval_a a2 s)
  | Sub (a1, a2) -> (eval_a a1 s) - (eval_a a2 s)
  | Mul (a1, a2) -> (eval_a a1 s) * (eval_a a2 s)
  | Div (a1, a2) -> (eval_a a1 s) / (eval_a a2 s)
  | Mod (a1, a2) -> (eval_a a1 s) mod (eval_a a2 s)
```

```
let rec eval_b : bexp -> State.t -> bool
=fun b s ->
  match b with
  | Bool true -> true
  | Bool false -> false
  | Eq (a1, a2) -> (eval_a a1 s) = (eval_a a2 s)
  | Le (a1, a2) -> (eval_a a1 s) <= (eval_a a2 s)
  | Neg b' -> not (eval_b b' s)
  | Conj (b1, b2) -> (eval_b b1 s) && (eval_b b2 s)

let rec eval_c : cmd -> State.t -> State.t
=fun c s ->
  match c with
  | Assign (x, a) -> State.update s x (eval_a a s)
  | Skip -> s
  | Seq (c1, c2) -> eval_c c2 (eval_c c1 s)
  | If (b, c1, c2) -> eval_c (if eval_b b s then c1 else c2) s
  | While (b, c) ->
    if eval_b b s then eval_c (While (b,c)) (eval_c c s)
    else s
  | Print a -> print_endline (string_of_int (eval_a a s)); s

let eval : program -> State.t
=fun p -> eval_c p State.empty
```

main.ml

```
let main () =
  let in_c =
    if Array.length Sys.argv != 2 then
      raise (Failure "No input is given")
    else
      Stdlib.open_in (Sys.argv.(1)) in
  let lexbuf = Lexing.from_channel in_c in
  let ast = Parser.program Lexer.next_token lexbuf in
    Eval.eval ast

let _ =  main ()
```

Summary

Implemented simple languages using a parser generator:

- `lexer.mll`, `parser.mly`: concrete syntax
- `ast.ml`: abstract syntax
- `eval.ml`: semantics (evaluation rules)