Principia Softwarica: (`ocaml`)`lex` and (`ocaml`)`yacc`
version 0.1

Yoann Padioleau
yoann.padioleau@gmail.com

with code from
Xavier Leroy and Yoann Padioleau

October 13, 2025

Contents

Chapter 1

Introduction

The goal of this book is to present in full details the source code of the lexer and parser code generators Lex and Yacc.

1.1 Motivations

Why Lex and Yacc? Because I think you are a better programmer if you fully understand how things work under the hood, and Lex and Yacc are used by many other programs such as the C compiler, the assembler, or even the shell.

1.2 `ocamllex` and `ocamlyacc`

1.3 Other compiler generators

Here are a few compiler generators that I considered for this book, but which I ultimately discarded:

- Unix Lex and Yacc
- GNU Flex and Bison
- Antlr
- Ometa

1.4 Getting started

1.5 Requirements

1.6 About this document

This document is a *literate program* [1]. It derives from a set of files processed by a tool, `syncweb` [2], generating either this book or the actual source code of the program. The code and its documentation are thus strongly connected.

1.7 Copyright

Most of this document is actually source code from OCaml, so those parts are copyright by INRIA. The prose is mine and is licensed under the GNU Free Documentation License.

⟨*copyright ocamllex* 8a⟩≡ (90d 88b 86 84 81 80 15a)

```
(***********************************************************************)
(*                                                                     *)
(*                             Objective Caml                          *)
(*                                                                     *)
(*            Xavier Leroy, projet Cristal, INRIA Rocquencourt         *)
(*                                                                     *)
(*  Copyright 1996 Institut National de Recherche en Informatique et   *)
(*  Automatique.  Distributed only by permission.                      *)
(*                                                                     *)
(***********************************************************************)
```

⟨*copyright ocamlyacc* 8b⟩≡ (100a 97b 96 94b 93b 92b 91 45a 42a)

```
(* Yoann Padioleau
 *
 * Copyright (C) 2015 Yoann Padioleau
 *
 * This library is free software; you can redistribute it and/or
 * modify it under the terms of the GNU Lesser General Public License
 * version 2.1 as published by the Free Software Foundation, with the
 * special exception on linking described in file license.txt.
 *
 * This library is distributed in the hope that it will be useful, but
 * WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the file
 * license.txt for more details.
 *)
```

1.8 Acknowledgments

Chapter 2

Overview

2.1 Lex and Yacc principles

2.2 `lex` command-line interface

⟨`Main.main()` *print lex usage if wrong number of arguments* 9⟩≡ (13b)

```
if Array.length Sys.argv != 2 then begin
  prerr_endline "Usage: ocamllex <input file>";
  exit 2
end;
```

2.3 `yacc` command-line interface

2.4 Toy calculator example

2.5 Code organization

2.6 Software architecture

2.7 Bootstrapping

2.8 Book structure

Part I

Lex

Chapter 3

Core data structures

3.1 Regular expression

⟨*type* Syntax.regular_expression 11a⟩≡ (80a)
```
type regular_expression =
    Epsilon
  | Characters of char_ list
  | Sequence of regular_expression * regular_expression
  | Alternative of regular_expression * regular_expression
  | Repetition of regular_expression
```

⟨*type* Syntax.char_ 11b⟩≡ (80a)
```
type char_ = int
```

3.2 Abstract syntax tree

⟨*type* Syntax.lexer_definition 11c⟩≡ (80a)
```
type lexer_definition =
    { header: location;
      entrypoints: rule list;
      trailer: location
    }
```

⟨*type* Syntax.location 11d⟩≡ (80a)
```
type location =
    Location of charpos * charpos
```

⟨*type* Syntax.charpos 11e⟩≡ (80a)
```
type charpos = int
```

⟨*type* Syntax.rule 11f⟩≡ (80a)
```
type rule = string * (regular_expression * action) list
```

⟨*type* Syntax.action 11g⟩≡ (80a)
```
type action = location
```

3.3 Automata

⟨*type* Lexgen.automata_entry 12a⟩≡ (84)
```
(* Representation of entry points *)

type automata_entry =
  { auto_name: string;
    auto_initial_state: int;
    auto_actions: (action_id * Ast.action) list;
  }
```

⟨*type* Lexgen.action_id 12b⟩≡ (84)
```
type action_id = int
```

⟨*type* Lexgen.automata_matrix 12c⟩≡ (84)
```
(* indexed by state number *)
type automata_matrix = automata_row array
```

⟨*type* Lexgen.automata 12d⟩≡ (84)
```
type automata_row =
    Perform of action_id
  (* indexed by an integer between 0 and 256(eof), that is a char_ *)
  | Shift of automata_trans * automata_move array
```

⟨*type* Lexgen.automata_trans 12e⟩≡ (84)
```
and automata_trans =
    No_remember
  | Remember of action_id
```

⟨*type* Lexgen.automata_move 12f⟩≡ (84)
```
and automata_move =
    Backtrack
  | Goto of int
```

3.4 Runtime `lexbuf`

⟨*type* Lexing.lexbuf 12g⟩≡ (90d 88b)
```
(* The run-time library for lexers generated by ocamllex *)
(* The type of lexer buffers. A lexer buffer is the argument passed
   to the scanning functions defined by the generated scanners.
   The lexer buffer holds the current state of the scanner, plus
   a function to refill the buffer from the input. *)
type lexbuf =
  { refill_buff : lexbuf -> unit;

    mutable lex_buffer : bytes;
    mutable lex_buffer_len : int;

    mutable lex_abs_pos : int;
    mutable lex_start_pos : int;
    mutable lex_curr_pos : int;

    mutable lex_last_pos : int;
    mutable lex_last_action : int;

    mutable lex_eof_reached : bool;

    (* used by lexers generated using the simpler code generation method *)
    mutable lex_last_action_simple : lexbuf -> Obj.t;
  }
```

Chapter 4

main()

⟨*toplevel* Main._1 13a⟩≡ (86c)
```
let _ =
  (* Printexc.catch*) main ();
  exit 0
```

⟨*function* Main.main 13b⟩≡ (86c)
```
let main () =
  ⟨Main.main() print lex usage if wrong number of arguments 9⟩
  let source_name = Sys.argv.(1) in
  let dest_name =
    if Filename.check_suffix source_name ".mll"
    then Filename.chop_suffix source_name ".mll" ^ ".ml"
    else source_name ^ ".ml"
  in
  let ic = open_in source_name in
  let oc = open_out dest_name in
  let lexbuf = Lexing.from_channel ic in

  (* parsing *)
  let def =
    try
      Parser.lexer_definition Lexer.main lexbuf
    with exn ->
      close_out oc;
      Sys.remove dest_name;
       ⟨Main.main() report error exn 78a⟩
      exit 2
  in
  (* compiling *)
  let (entries, transitions) = Lexgen.make_dfa def in

(* CONFIG
  Output.output_lexdef_simple ic oc
    def.header (entries, transitions) def.trailer;
*)
  (* optimizing *)
  let tables = Compact.compact_tables transitions in
  (* generating *)
  Output.output_lexdef ic oc def.header tables entries def.trailer;
  close_in ic;
  close_out oc
```

⟨*signature* Lexgen.make_dfa 13c⟩≡ (84a)
```
(* The entry point *)
```

```
  val make_dfa: Ast.lexer_definition -> automata_entry list * automata_matrix
```

⟨*signature* Output.output_lexdef 14⟩≡ (80b)

```
  (* Output the DFA tables and its entry points *)

  val output_lexdef:
        in_channel -> out_channel ->
        Ast.location (* header *) ->
        Compact.lex_tables ->
        Lexgen.automata_entry list ->
        Ast.location (* trailer *) ->
        unit
```

Chapter 5

Lexing

⟨lex/lexer.mll 15a⟩≡
```
⟨copyright ocamllex 8a⟩
(* The lexical analyzer for lexer definitions. Bootstrapped! *)

{
open Stdcompat (* for Bytes *)
open Ast
open Parser

exception Lexical_error of string

(* Auxiliaries for the lexical analyzer *)
⟨Lexer helper functions and globals 16c⟩
}

⟨rule Lexer.main 15c⟩

⟨rule Lexer.action 18g⟩

⟨rule Lexer.string 17i⟩

⟨rule Lexer.comment 16d⟩
```

⟨type Parser.token 15b⟩≡ (20a)
```
%token Trule Tparse Tand
%token <int> Tchar
%token <string> Tstring
%token Tstar Tmaybe Tplus Tor Tlparen Trparen
%token Tlbracket Trbracket Tcaret Tdash
%token Tunderscore Teof
%token <Ast.location> Taction
%token Tlet Tequal
%token <string> Tident
%token Tend
```

⟨rule Lexer.main 15c⟩≡ (15a)
```
rule main = parse
  ⟨Lexer.main() space case 16a⟩
  ⟨Lexer.main() comment case 16b⟩
  ⟨Lexer.main() keyword or identifier case 16e⟩
  ⟨Lexer.main() string start case 17e⟩
  ⟨Lexer.main() character cases 18d⟩
  ⟨Lexer.main() operator cases 16f⟩
  ⟨Lexer.main() action case 18e⟩
  | eof  { Tend }
```

```
  | _
    { raise(Lexical_error
              ("illegal character " ^ String.escaped(Lexing.lexeme lexbuf))) }
```

5.1 Comments

⟨Lexer.main() *space case* 16a⟩≡ (15c)
```
    [' ' '\010' '\013' '\009' '\012' ] +
    { main lexbuf }
```

⟨Lexer.main() *comment case* 16b⟩≡ (15c)
```
  | "(*"
    { comment_depth := 1;
      comment lexbuf;
      main lexbuf }
```

⟨*Lexer helper functions and globals* 16c⟩≡ (15a) 17f▷
```
  let comment_depth = ref 0
```

⟨*rule Lexer.comment* 16d⟩≡ (15a)
```
  and comment = parse
      "(*"
      { incr comment_depth; comment lexbuf }
    | "*)"
      { decr comment_depth;
        if !comment_depth == 0 then () else comment lexbuf }

    | eof
      { raise(Lexical_error "unterminated comment") }
    | _
      { comment lexbuf }
```

5.2 Keywords and identifiers

⟨Lexer.main() *keyword or identifier case* 16e⟩≡ (15c)
```
  | ['A'-'Z' 'a'-'z'] ['A'-'Z' 'a'-'z' '\'' '_' '0'-'9'] *
    { match Lexing.lexeme lexbuf with
      | "rule" -> Trule
      | "parse" -> Tparse
      | "and" -> Tand
      | "eof" -> Teof
      | "let" -> Tlet
      | s -> Tident s
     }
```

5.3 Operators

⟨Lexer.main() *operator cases* 16f⟩≡ (15c) 16g▷
```
  | '*'  { Tstar }
  | '|'  { Tor }
```

⟨Lexer.main() *operator cases* 16g⟩+≡ (15c) ◁16f 17a▷
```
  | '?'  { Tmaybe }
  | '+'  { Tplus }
```

⟨Lexer.main() *operator cases* 17a⟩+≡ (15c) ◁16g 17b▷
```
  | '('  { Tlparen }
  | ')'  { Trparen }
```

⟨Lexer.main() *operator cases* 17b⟩+≡ (15c) ◁17a 17c▷
```
  | '['  { Tlbracket }
  | ']'  { Trbracket }
  | '-'  { Tdash }
  | '^'  { Tcaret }
```

⟨Lexer.main() *operator cases* 17c⟩+≡ (15c) ◁17b 17d▷
```
  | '_'  { Tunderscore }
```

⟨Lexer.main() *operator cases* 17d⟩+≡ (15c) ◁17c
```
  | '='  { Tequal }
```

5.4 Strings

⟨Lexer.main() *string start case* 17e⟩≡ (15c)
```
  | '"'
    { reset_string_buffer();
      string lexbuf;
      Tstring(get_stored_string()) }
```

⟨*Lexer helper functions and globals* 17f⟩+≡ (15a) ◁16c 17g▷
```
  let initial_string_buffer = Bytes.create 256
  let string_buff = ref initial_string_buffer
  let string_index = ref 0
```

⟨*Lexer helper functions and globals* 17g⟩+≡ (15a) ◁17f 17h▷
```
  let reset_string_buffer () =
    string_buff := initial_string_buffer;
    string_index := 0
```

⟨*Lexer helper functions and globals* 17h⟩+≡ (15a) ◁17g 18a▷
```
  let get_stored_string () =
    Bytes.sub_string !string_buff 0 !string_index
```

⟨*rule Lexer.string* 17i⟩≡ (15a)
```
  and string = parse
      '"'
      { () }
    | '\\' [' ' '\010' '\013' '\009' '\026' '\012'] +
      { string lexbuf }
    | '\\' ['\\' '"' 'n' 't' 'b' 'r']
      { store_string_char(char_for_backslash(Lexing.lexeme_char lexbuf 1));
        string lexbuf }
    | '\\' ['0'-'9'] ['0'-'9'] ['0'-'9']
      { store_string_char(char_for_decimal_code lexbuf 1);
        string lexbuf }
    | eof
      { raise(Lexical_error "unterminated string") }
    | _
      { store_string_char(Lexing.lexeme_char lexbuf 0);
        string lexbuf }
```

⟨*Lexer helper functions and globals* 18a⟩+≡ (15a) ◁17h 18b▷
```
let store_string_char c =
  if !string_index >= Bytes.length !string_buff then begin
    let new_buff = Bytes.create (Bytes.length !string_buff * 2) in
    Bytes.blit !string_buff 0 new_buff 0 (Bytes.length !string_buff);
    string_buff := new_buff
  end;
  Bytes.set !string_buff !string_index c;
  incr string_index
```

⟨*Lexer helper functions and globals* 18b⟩+≡ (15a) ◁18a 18c▷
```
let char_for_backslash = function
    'n' -> '\n'
  | 't' -> '\t'
  | 'b' -> '\b'
  | 'r' -> '\r'
  | c   -> c
```

⟨*Lexer helper functions and globals* 18c⟩+≡ (15a) ◁18b 18f▷
```
let char_for_decimal_code lexbuf i =
  Char.chr(100 * (Char.code(Lexing.lexeme_char lexbuf i) - 48) +
           10 * (Char.code(Lexing.lexeme_char lexbuf (i+1)) - 48) +
                (Char.code(Lexing.lexeme_char lexbuf (i+2)) - 48))
```

5.5 Characters

⟨Lexer.main() *character cases* 18d⟩≡ (15c)
```
| "'" [^ '\\'] "'"
  { Tchar(Char.code(Lexing.lexeme_char lexbuf 1)) }
| "'" '\\' ['\\' '\'' 'n' 't' 'b' 'r'] "'"
  { Tchar(Char.code(char_for_backslash (Lexing.lexeme_char lexbuf 2))) }
| "'" '\\' ['0'-'9'] ['0'-'9'] ['0'-'9'] "'"
  { Tchar(Char.code(char_for_decimal_code lexbuf 2)) }
```

5.6 Actions

⟨Lexer.main() *action case* 18e⟩≡ (15c)
```
| '{'
  { let n1 = Lexing.lexeme_end lexbuf in
    brace_depth := 1;
    let n2 = action lexbuf in
    Taction(Location(n1, n2)) }
```

⟨*Lexer helper functions and globals* 18f⟩+≡ (15a) ◁18c
```
let brace_depth = ref 0
```

⟨*rule Lexer.action* 18g⟩≡ (15a)
```
and action = parse
    '{'
    { incr brace_depth;
      action lexbuf }
  | '}'
    { decr brace_depth;
      if !brace_depth == 0 then Lexing.lexeme_start lexbuf else action lexbuf }
  | "(*"
    { comment_depth := 1;
```

```
    comment lexbuf;
    action lexbuf }
| eof
  { raise (Lexical_error "unterminated action") }
| _
  { action lexbuf }
```

Chapter 6

Parsing

6.1 Overview

⟨lex/parser.mly 20a⟩≡
⟨*copyright ocamllex bis* 87a⟩

```
/* The grammar for lexer definitions */

%{
open Ast

(* Auxiliaries for the parser. *)
```
⟨*parser helper functions and globals* 22f⟩
```
%}

/* Tokens */
```
⟨*type* Parser.token 15b⟩
```
/* Precedences and associativities. Lower precedences come first. */
```
⟨*Parser precedences and associativities* 22a⟩
```
/* Entry points */
```
⟨*Parser entry points types* 21a⟩
```
%%
```
⟨*grammar(lex)* 20b⟩
```
%%
```

⟨*grammar(lex)* 20b⟩≡ (20a)
⟨*lex top rule* 21b⟩

⟨*lex header rule* 21c⟩

⟨*lex named regexp rule* 23e⟩

⟨*lex rule rule* 21d⟩

⟨*lex regexp rule* 22b⟩

6.2 Lexer definition entry point

⟨*Parser entry points types* 21a⟩≡ (20a)
```
%start lexer_definition
%type <Ast.lexer_definition> lexer_definition
```

⟨*lex top rule* 21b⟩≡ (20b)
```
lexer_definition:
    header named_regexps Trule definition other_definitions header Tend
        { { header = $1;
            entrypoints = $4 :: List.rev $5;
            trailer = $6
          }
         }
;
```

6.3 Header and trailer

⟨*lex header rule* 21c⟩≡ (20b)
```
header:
    Taction      { $1 }
  | /*epsilon*/  { Location(0,0) }
;
```

6.4 Lexing rule

⟨*lex rule rule* 21d⟩≡ (20b)
```
definition:
    Tident Tequal entry
        { ($1,$3) }
;
entry:
    Tparse case rest_of_entry
        { $2::List.rev $3 }
  | Tparse rest_of_entry
        { List.rev $2 }
;
case:
    regexp Taction
        { ($1,$2) }
;


other_definitions:
    other_definitions Tand definition
        { $3::$1 }
  | /*epsilon*/
        { [] }
;
rest_of_entry:
    rest_of_entry Tor case
        { $3::$1 }
  | /*epsilon*/
        { [] }
;
```

6.5 Regexps

6.5.1 Basic regexps

⟨*Parser precedences and associativities* 22a⟩≡ (20a)
```
%left Tor
%left CONCAT
%nonassoc Tmaybe
%left Tstar
%left Tplus
```

⟨*lex regexp rule* 22b⟩≡ (20b) 22e▷
```
regexp:
    Tchar
        { Characters [$1] }
  | regexp Tstar
        { Repetition $1 }
  | regexp Tor regexp
        { Alternative($1,$3) }
  | regexp regexp %prec CONCAT
        { Sequence($1,$2) }
  | Tlparen regexp Trparen
        { $2 }
  ⟨rule regexp cases 22c⟩
;
```

6.5.2 Sugar regexps

⟨*rule regexp cases* 22c⟩≡ (22b) 22d▷
```
  | regexp Tmaybe
        { Alternative($1, Epsilon) }
  | regexp Tplus
        { Sequence($1, Repetition $1) }
```

6.5.3 Range regexps

⟨*rule regexp cases* 22d⟩+≡ (22b) ◁22c 23d▷
```
  | Tlbracket char_class Trbracket
        { Characters $2 }
```

⟨*lex regexp rule* 22e⟩+≡ (20b) ◁22b
```
char_class:
    Tcaret char_class1
        { subtract all_chars $2 }
  | char_class1
        { $1 }
;
char_class1:
    Tchar Tdash Tchar
        { char_class $1 $3 }
  | Tchar
        { [$1] }
  | char_class1 char_class1 %prec CONCAT
        { $1 @ $2 }
;
```

⟨*parser helper functions and globals* 22f⟩≡ (20a) 23a▷

⟨*parser helper functions and globals* 23a⟩+≡ (20a) ◁22f 23b▷

⟨*parser helper functions and globals* 23b⟩+≡ (20a) ◁23a 23c▷
```
let rec subtract l1 l2 =
  match l1 with
    [] -> []
  | a::r -> if List.mem a l2 then subtract r l2 else a :: subtract r l2
```

6.5.4 Named regexps

⟨*parser helper functions and globals* 23c⟩+≡ (20a) ◁23b 23g▷
```
let named_regexps =
  (Hashtbl.create 13 : (string, regular_expression) Hashtbl.t)
```

⟨*rule regexp cases* 23d⟩+≡ (22b) ◁22d 23f▷
```
  | Tident
        { try
            Hashtbl.find named_regexps $1
          with Not_found ->
            prerr_string "Reference to unbound regexp name `";
            prerr_string $1;
            prerr_string "' at char ";
            prerr_int (Parsing.symbol_start());
            prerr_newline();
            exit 2 }
```

⟨*lex named regexp rule* 23e⟩≡ (20b)
```
named_regexps:
    named_regexps Tlet Tident Tequal regexp
        { Hashtbl.add named_regexps $3 $5 }
  | /*epsilon*/
        { () }
;
```

6.5.5 Other regexps

⟨*rule regexp cases* 23f⟩+≡ (22b) ◁23d 23h▷
```
  | Tstring
        { regexp_for_string $1 }
```

⟨*parser helper functions and globals* 23g⟩+≡ (20a) ◁23c
```
let regexp_for_string s =
  let rec re_string n =
    if n >= String.length s then Epsilon
    else if succ n = String.length s then Characters([Char.code (s.[n])])
    else Sequence(Characters([Char.code (s.[n])]), re_string (succ n))
  in re_string 0
```

⟨*rule regexp cases* 23h⟩+≡ (22b) ◁23f
```
  | Tunderscore
        { Characters all_chars }
  | Teof
        { Characters [Ast.char_eof] }
```

Chapter 7

Checking

Chapter 8

Compiling

⟨*function* Lexgen.make_dfa 25a⟩≡ (84b)
```
let make_dfa lexdef =
  let (charsets, lex_entries) =
    encode_lexdef lexdef in
  let (automata_entries, automata_transitions) =
    encode_lexentries charsets lex_entries in
  automata_entries, automata_transitions
```

8.1 Normalized regexps

8.1.1 Intermediate regexp format

⟨*type* Lexgen.regexp 25b⟩≡ (84b)
```
(* Deep abstract syntax for regular expressions *)

type regexp =
    Empty
  | Chars of charset_id
  | Action of action_id
  | Seq of regexp * regexp
  | Alt of regexp * regexp
  | Star of regexp
```

⟨*type* Lexgen.lexer_entry 25c⟩≡ (84b)
```
type lexer_entry =
  { lex_name: string;
    lex_regexp: regexp;
    lex_actions: (action_id * Ast.action) list;
  }
```

⟨*type* Lexgen.charset_id 25d⟩≡ (84b)
```
type charset_id = int
```

⟨*constant* Lexgen.chars 25e⟩≡ (84b)
```
let chars = ref ([] : char_ list list)
```

⟨*constant* Lexgen.chars_count 25f⟩≡ (84b)
```
let chars_count = ref (0: charset_id)
```

⟨*constant* Lexgen.actions 25g⟩≡ (84b)
```
let actions = ref ([] : (action_id * Ast.location) list)
```

⟨*constant* Lexgen.actions_count 25h⟩≡ (84b)
```
let actions_count = ref (0: action_id)
```

8.1.2 encode_lexdef()

⟨*function* Lexgen.encode_lexdef 26a⟩≡ (84b)
```
let encode_lexdef def =
  chars := [];
  chars_count := 0;
(* CONFIG   actions_count := 0; *)
  let entries =
    def.entrypoints |> List.map (fun (entry_name, casedef) ->
        actions := [];
        (* CONFIG !! for simpler output can't do that *)
        actions_count := 0;
        let re = encode_casedef casedef in
        { lex_name = entry_name;
          lex_regexp = re;
          lex_actions = List.rev !actions }
    )
  in
  (* map a charset_id to a charset *)
  let charsets = Array.of_list (List.rev !chars) in
  (charsets, entries)
```

8.1.3 encode_casedef()

⟨*function* Lexgen.encode_casedef 26b⟩≡ (84b)
```
let encode_casedef casedef =
  casedef |> List.fold_left (fun reg (re, action) ->
     let act_num = !actions_count in
     incr actions_count;
     actions := (act_num, action) :: !actions;
     Alt(reg, Seq(encode_regexp re, Action act_num))
  ) Empty
```

8.1.4 encode_regexp()

⟨*function* Lexgen.encode_regexp 26c⟩≡ (84b)
```
let rec encode_regexp = function
    Epsilon -> Empty
  | Characters cl ->
      let n = !chars_count in
      incr chars_count;
      chars := cl :: !chars;
      Chars(n)
  | Sequence(r1,r2) ->
      Seq(encode_regexp r1, encode_regexp r2)
  | Alternative(r1,r2) ->
      Alt(encode_regexp r1, encode_regexp r2)
  | Repetition r ->
      Star (encode_regexp r)
```

8.2 DFA

⟨*type* Lexgen.transition 26d⟩≡ (84b)
```
type transition =
    OnChars of charset_id
  | ToAction of action_id
```

⟨*type* Lexgen.state 27a⟩≡ (84b)
```
type state = transition Set.t
```

⟨*function* Lexgen.reset_state_mem 27b⟩≡ (84b)
```
let reset_state_mem () =
  state_map := Map.empty;
  Stack.clear todo;
  next_state_num := 0
```

⟨*constant* Lexgen.state_map 27c⟩≡ (84b)
```
let state_map = ref (Map.empty: (state, int) Map.t)
```

⟨*constant* Lexgen.todo 27d⟩≡ (84b)
```
let todo = (Stack.create() : (state * int) Stack.t)
```

⟨*constant* Lexgen.next_state_num 27e⟩≡ (84b)
```
let next_state_num = ref 0
```

⟨*function* Lexgen.encode_lexentries 27f⟩≡ (84b)
```
let encode_lexentries charsets lexentries =
  reset_state_mem();

  (* pass 1 on lexentries, get the initial states *)
  let automata_entries =
    lexentries |> List.map (fun entry ->
        { auto_name    = entry.lex_name;
          auto_actions = entry.lex_actions;
          (* get_state() will populate todo and state_map globals *)
          auto_initial_state = get_state (firstpos entry.lex_regexp);
        }
    )
  in
  (* pass 2 on lexentries, get the transitions *)

  let follow = followpos (Array.length charsets) lexentries in
  let states =
    map_on_all_states (fun st -> translate_state charsets follow st)
  in
  let transitions = Array.make !next_state_num (Perform 0) in
  states |> List.iter (fun (act, i) -> transitions.(i) <- act);
  (automata_entries, transitions)
```

8.2.1 Initial states

⟨*function* Lexgen.firstpos 27g⟩≡ (84b)
```
let rec firstpos = function
    Empty      -> Set.empty
  | Chars pos  -> Set.add (OnChars pos) Set.empty
  | Action act -> Set.add (ToAction act) Set.empty
  | Seq(r1,r2) -> if nullable r1
                  then Set.union (firstpos r1) (firstpos r2)
                  else firstpos r1
  | Alt(r1,r2) -> Set.union (firstpos r1) (firstpos r2)
  | Star r     -> firstpos r
```

⟨*function* Lexgen.nullable 28a⟩≡ (84b)
```
let rec nullable = function
    Empty      -> true
  | Chars _    -> false
  | Action _   -> false
  | Seq(r1,r2) -> nullable r1 && nullable r2
  | Alt(r1,r2) -> nullable r1 || nullable r2
  | Star _r     -> true
```

⟨*function* Lexgen.get_state 28b⟩≡ (84b)
```
let get_state st =
  try
    Map.find st !state_map
  with Not_found ->
    let num = !next_state_num in
    incr next_state_num;
    state_map := Map.add st num !state_map;
    Stack.push (st, num) todo;
    num
```

8.2.2 Transitions

⟨*function* Lexgen.map_on_all_states 28c⟩≡ (84b)
```
let map_on_all_states f =
  let res = ref [] in
  begin try
    while true do
      let (st, i) = Stack.pop todo in
      let r = f st in
      res := (r, i) :: !res
    done
  with Stack.Empty -> ()
  end;
  !res
```

⟨*function* Lexgen.translate_state 28d⟩≡ (84b)
```
let translate_state charsets follow =
 fun state ->
  match split_trans_set state with
    (n, []) -> Perform n
  | (n, ps) -> Shift((if n = no_action then No_remember else Remember n),
                     transition_from charsets follow ps)
```

⟨*function* Lexgen.split_trans_set 28e⟩≡ (84b)
```
let split_trans_set trans_set =
  Set.fold (fun trans (act, pos_set as act_pos_set) ->
      match trans with
        OnChars pos -> (act, pos :: pos_set)
      | ToAction act1 -> if act1 < act then (act1, pos_set) else act_pos_set
   ) trans_set (no_action, [])
```

⟨*constant* Lexgen.no_action 28f⟩≡ (84b)
```
let no_action = (max_int: action_id)
```

⟨*function* Lexgen.transition_from 29a⟩≡ (84b)

```
let transition_from charsets follow pos_set =
  let tr    = Array.make (Ast.charset_size + 1) Set.empty in
  pos_set |> List.iter (fun pos ->
     charsets.(pos) |> List.iter (fun c ->
           tr.(c) <- Set.union tr.(c) follow.(pos)
      )
  );

  let shift = Array.make (Ast.charset_size + 1) Backtrack in
  for i = 0 to Ast.charset_size do
    shift.(i) <- goto_state tr.(i)
  done;
  shift
```

⟨*function* Lexgen.goto_state 29b⟩≡ (84b)

```
let goto_state st =
  if Set.is_empty st
  then Backtrack
  (* can create a new state in todo *)
  else Goto (get_state st)
```

⟨*function* Lexgen.followpos 29c⟩≡ (84b)

```
let followpos size_charsets entries =
  let v = Array.make size_charsets Set.empty in
  let fill_pos first = function
      OnChars pos -> v.(pos) <- Set.union first v.(pos)
    | ToAction _  -> ()
  in
  let rec fill = function
      Seq(r1,r2) ->
        fill r1; fill r2;
        Set.iter (fill_pos (firstpos r2)) (lastpos r1)
    | Alt(r1,r2) ->
        fill r1; fill r2
    | Star r ->
        fill r;
        Set.iter (fill_pos (firstpos r)) (lastpos r)
    | _ -> () in
  entries |> List.iter (fun entry -> fill entry.lex_regexp);
  v
```

⟨*function* Lexgen.lastpos 29d⟩≡ (84b)

```
let rec lastpos = function
    Empty      -> Set.empty
  | Chars pos  -> Set.add (OnChars pos) Set.empty
  | Action act -> Set.add (ToAction act) Set.empty
  | Seq(r1,r2) -> if nullable r2
                  then Set.union (lastpos r1) (lastpos r2)
                  else lastpos r2
  | Alt(r1,r2) -> Set.union (lastpos r1) (lastpos r2)
  | Star r     -> lastpos r
```

Chapter 9

Generating

⟨*function* Output.output_lexdef 30a⟩≡ (81)
```
(* Main output function *)

let output_lexdef ic oc header tables entry_points trailer =
  ⟨Output.output_lexdef() print statistics 30b⟩
  copy_chunk ic oc header;
  output_tables oc tables;
  ⟨Output.output_lexdef() generate entry points 30c⟩
  copy_chunk ic oc trailer
```

⟨Output.output_lexdef() *print statistics* 30b⟩≡ (30a)
```
Printf.printf "%d states, %d transitions, table size %d bytes\n"
  (Array.length tables.tbl_base)
  (Array.length tables.tbl_trans)
  (2 * (Array.length tables.tbl_base + Array.length tables.tbl_backtrk +
        Array.length tables.tbl_default + Array.length tables.tbl_trans +
        Array.length tables.tbl_check));
flush stdout;
```

9.1 Entry points

⟨Output.output_lexdef() *generate entry points* 30c⟩≡ (30a)
```
(match entry_points with
  [] -> ()
| entry1 :: entries ->
    output_string oc "let rec ";
    output_entry ic oc entry1;
    entries |> List.iter (fun e ->
      output_string oc "and ";
      output_entry ic oc e
    )
);
```

⟨*function* Output.output_entry 30d⟩≡ (81)
```
(* Output the entries *)

let output_entry ic oc e =
  fprintf oc "%s lexbuf = %s_rec lexbuf %d\n"
          e.auto_name e.auto_name e.auto_initial_state;
  fprintf oc "and %s_rec lexbuf state =\n" e.auto_name;
  fprintf oc "  match Lexing.engine lex_tables state lexbuf with\n    ";
  let first = ref true in
  e.auto_actions |> List.iter (fun (num, loc_action) ->
```

```
        if !first
        then first := false
        else fprintf oc "  | ";
        fprintf oc "%d -> (" num;
        copy_chunk ic oc loc_action;
        fprintf oc ")\n"
    );
    fprintf oc "  | n -> lexbuf.Lexing.refill_buff lexbuf; %s_rec lexbuf n\n\n"
            e.auto_name
```

9.2 Shared transition table

⟨*function* `Output.output_tables` 31a⟩≡ (81)

```
  (* Output the tables *)

  let output_tables oc tbl =
    output_string oc "let lex_tables = {\n";
    fprintf oc "  Lexing.lex_base = \n%a;\n" output_array tbl.tbl_base;
    fprintf oc "  Lexing.lex_backtrk = \n%a;\n" output_array tbl.tbl_backtrk;
    fprintf oc "  Lexing.lex_default = \n%a;\n" output_array tbl.tbl_default;
    fprintf oc "  Lexing.lex_trans = \n%a;\n" output_array tbl.tbl_trans;
    fprintf oc "  Lexing.lex_check = \n%a\n" output_array tbl.tbl_check;
    output_string oc "}\n\n"
```

⟨*function* `Output.output_array` 31b⟩≡ (81)

```
  let output_array oc v =
    output_string oc "   \"";
    for i = 0 to Array.length v - 1 do
      output_byte oc (v.(i) land 0xFF);
      output_byte oc ((v.(i) asr 8) land 0xFF);
      if i land 7 = 7 then output_string oc "\\\n    "
    done;
    output_string oc "\""
```

⟨*function* `Output.output_byte` 31c⟩≡ (81)

```
  (* To output an array of short ints, encoded as a string *)

  let output_byte oc b =
    output_char oc '\\';
    output_char oc (Char.chr(48 + b / 100));
    output_char oc (Char.chr(48 + (b / 10) mod 10));
    output_char oc (Char.chr(48 + b mod 10))
```

9.3 Chunks

⟨*function* `Output.copy_chunk` 31d⟩≡ (81)

```
  let copy_chunk ic oc (Location(start,stop)) =
    seek_in ic start;
    let n = ref (stop - start) in
    while !n > 0 do
      let m = input ic copy_buffer 0 (min !n 1024) in
      output oc copy_buffer 0 m;
      n := !n - m
    done
```

⟨*constant* `Output.copy_buffer` 32⟩≡ (81)

```
(* To copy the ML code fragments *)

let copy_buffer = Bytes.create 1024
```

Chapter 10

Running

Part II

Yacc

Chapter 11

Core data structures

11.1 Context free grammar

⟨*type* `Ast.grammar` *(yacc)* 35a⟩≡ (91)
```
type grammar = rule list
```

⟨*type* `Ast.rule_` *(yacc)* 35b⟩≡ (91)
```
    and rule = {
      lhs: nonterm;
      rhs: symbol list;
      act: action;
    }
```

⟨*type* `Ast.term` *(yacc)* 35c⟩≡ (91)
```
(* uppercase string *)
type term = T of string
```

⟨*type* `Ast.nonterm` *(yacc)* 35d⟩≡ (91)
```
(* lowercase string *)
type nonterm = NT of string
```

⟨*type* `Ast.symbol` *(yacc)* 35e⟩≡ (91)
```
type symbol = Term of term | Nonterm of nonterm
```

⟨*type* `Ast.charpos` *(yacc)* 35f⟩≡ (91)
```
type charpos = int
```

⟨*type* `Ast.location` *(yacc)* 35g⟩≡ (91)
```
type location =
    Location of charpos * charpos
```

⟨*type* `Ast.action` *(yacc)* 35h⟩≡ (91)
```
(* the slice may contain the special $<digit> markers *)
type action = location
```

⟨*constant* `Ast.noloc` *(yacc)* 35i⟩≡ (91)
```
let noloc = Location(0, 0)
```

⟨*constant* `Tests.arith`*(yacc)* 36a⟩≡ (98)

```
(* from tests/yacc/arith.mly which is a copy of the representative grammar in
 * the dragon book in 4.1
 * $S -> E         (R0)
 * E -> E + T | T  (R1, R2)
 * T -> T * F | F  (R3, R4)
 * F -> ( E ) | id (R5, R6)
 *)
let arith =
    [{lhs = NT "e";
      rhs = [Nonterm (NT "e"); Term (T "PLUS");  Nonterm (NT "t")];
      act = noloc};
     {lhs = NT "e";
      rhs = [Nonterm (NT "t")];
      act = noloc};
     {lhs = NT "t";
      rhs = [Nonterm (NT "t"); Term (T "MULT"); Nonterm (NT "f")];
      act = noloc};
     {lhs = NT "t"; rhs = [Nonterm (NT "f")];
      act = noloc};
     {lhs = NT "f";
      rhs = [Term (T "TOPAR"); Nonterm (NT "e"); Term (T "TCPAR")];
      act = noloc};
     {lhs = NT "f";
      rhs = [Term (T "ID")];
      act = noloc}]
(*
let augmented_arith =
  {lhs = NT "$S"; rhs = [Nonterm (NT "e")]; act = noloc} :: arith
*)
```

11.2 Abstract syntax tree

⟨*type* `Ast.parser_definition`*(yacc)* 36b⟩≡ (91)

```
(* main data structure *)
type parser_definition = {
  header: location;
  directives: directive list;
  grm: grammar;
  trailer: location;
}
```

⟨*type* `Ast.directive`*(yacc)* 36c⟩≡ (91)

```
type directive =
  | Token of type_ option * term
  | Start of nonterm
  | Type of type_ * nonterm
  | Prec of unit (* TODO *)
```

⟨*type* `Ast.type_`*(yacc)* 36d⟩≡ (91)

```
  and type_ = string
```

11.3 Pushdown automata

⟨*type* `Lr0.stateid`*(yacc)* 36e⟩≡ (93)

```
type stateid = S of int
```

⟨*type* `Lrtables.lr_tables`*(yacc)* 37a⟩≡ (95a)
```
type lr_tables = action_table * goto_table
```

⟨*type* `Lrtables.action_table`*(yacc)* 37b⟩≡ (95a)
```
(* term can be also the special "$" terminal.
 * Everything not in the list is an Error action, so this
 * list should not contain any Error.
 *)
type action_table =
    ((Lr0.stateid * Ast.term) * action) list
```

⟨*type* `Lrtables.action`*(yacc)* 37c⟩≡ (95a)
```
type action =
  | Shift of Lr0.stateid
  | Reduce of Lr0.ruleidx
  | Accept
  | Error
```

⟨*type* `Lrtables.goto_table`*(yacc)* 37d⟩≡ (95a)
```
type goto_table =
    ((Lr0.stateid * Ast.nonterm) * Lr0.stateid) list
```

⟨*type* `Lr0.ruleidx`*(yacc)* 37e⟩≡ (93)
```
(* the index of the rule in env.g *)
type ruleidx = R of int
```

⟨*type* `Lr0.env`*(yacc)* 37f⟩≡ (93)
```
type env = {
  (* augmented grammar where r0 is $S -> start_original_grammar *)
  g: Ast.rule array;
}
```

⟨*constant* `Ast.start_nonterminal`*(yacc)* 37g⟩≡ (91)
```
(* They should not conflict with user-defined terminals or non terminals
 * because nonterminals cannot contain '$' according to lexer.mll and
 * terminals must start with an uppercase letter according again
 * to lexer.mll
 *)
let start_nonterminal = NT "S$"
```

⟨*constant* `Ast.dollar_terminal`*(yacc)* 37h⟩≡ (91)
```
let dollar_terminal = T "$"
```

11.4 Runtime `lr_tables` and `parser_env`

⟨*type* `Parsing.lr_tables`*(yacc)* 37i⟩≡ (106 101h)
```
type 'tok lr_tables = {
  action: stateid * 'tok -> action;
  goto: stateid * nonterm -> stateid;
}
```

⟨*type* `Parsing.stateid`*(yacc)* 37j⟩≡ (106 101h)
```
type stateid = S of int
```

⟨*type* `Parsing.nonterm`*(yacc)* 37k⟩≡ (106 101h)
```
(* less: could be an index, but easier for debug to use the original name *)
type nonterm = NT of string
```

⟨*type* Parsing.action*(yacc)* 38a⟩≡ (106 101h)
```
type action =
  | Shift of stateid
  | Reduce of nonterm * int (* size of rhs of the rule *) * rule_action
  | Accept
```

⟨*exception* Parsing.Parse_error*(yacc)* 38b⟩≡ (101h)
```
exception Parse_error
        (* Raised when a parser encounters a syntax error.
           Can also be raised from the action part of a grammar rule,
           to initiate error recovery. *)
```

⟨*type* Parsing.rule_action*(yacc)* 38c⟩≡ (106 101h)
```
(* index in the rule actions table passed to yyparse *)
type rule_action = RA of int
```

⟨*type* Parsing.rules_actions*(yacc)* 38d⟩≡ (106 101h)
```
type rules_actions = (parser_env_simple -> Obj.t) array
```

⟨*signature* Parsing.yyparse_simple*(yacc)* 38e⟩≡ (101h)
```
val yyparse_simple:
  'tok lr_tables -> rules_actions ->
  (Lexing.lexbuf -> 'tok) -> ('tok -> string) -> Lexing.lexbuf -> 'a
```

⟨*type* Parsing.parser_env_simple*(yacc)* 38f⟩≡ (106)
```
type parser_env_simple = {
  states: stateid Stack.t;
  (* todo: opti: could use a growing array as one oftens needs to index it
   * with the peek_val
   * The semantic attributes (token value or non terminal value).
   *)
  values: Obj.t Stack.t;
  mutable current_rule_len: int;
}
```

⟨*type* Tests.token*(yacc)* 38g⟩≡ (98)
```
type token =
  | T0
  | TEOF
```

⟨*function* Tests.test_lr_engine*(yacc)* 38h⟩≡ (98)
```
(* what we should generate *)
let test_lr_engine () =
  let tokens = ref [T0; TEOF] in
  let lexbuf = Lexing.from_string "fake" in

  let lexfun _lexbuf =
    match !tokens with
    | [] -> failwith "no more tokens"
    | x::xs ->
      tokens := xs;
      x
  in
  let lrtables = {
    Parsing_.action = (function
      | (S 0, T0) -> Shift (S 1)
      | (S 1, TEOF) -> Reduce (NT "S", 1, RA 1)
      | (S 2, TEOF) -> Accept
      | _ -> raise Parsing.Parse_error
    );
```

```
  Parsing_.goto = (function
    | S 0, NT "S" -> S 2
    | _ -> raise Parsing_.Parse_error
  );
}
in
(* todo *)
let rules_action = [||] in
let string_of_tok = function | T0 -> "T0" | TEOF -> "TEOF" in

Parsing_.yyparse_simple lrtables rules_action lexfun string_of_tok lexbuf
```

Chapter 12

main()

⟨*function* Main.main*(yacc)* 40⟩≡ (100a)

```
let main () =

  if Array.length Sys.argv != 2 then begin
    prerr_endline "Usage: ocamlyacc <input file>";
    exit 2
  end;

  let source_name = Sys.argv.(1) in

  let dest_name =
    if Filename.check_suffix source_name ".mly"
    then Filename.chop_suffix source_name ".mly" ^ ".ml"
    else source_name ^ ".ml"
  in
  let ic = open_in source_name in
  let lexbuf = Lexing.from_channel ic in

  (* parsing *)
  let def =
    try
      Parser.parser_definition Lexer.main lexbuf
    with exn ->
      Sys.remove dest_name;
       (match exn with
         Parsing.Parse_error ->
           prerr_string "Syntax error around char ";
           prerr_int (Lexing.lexeme_start lexbuf);
           prerr_endline "."
       | Lexer.Lexical_error s ->
           prerr_string "Lexical error around char ";
           prerr_int (Lexing.lexeme_start lexbuf);
           prerr_string ": ";
           prerr_string s;
           prerr_endline "."
       | _ -> raise exn
       );
      exit 2
  in
  let env = Lr0.mk_env_augmented_grammar (Ast.start_symbol def) def.grm  in
  let automaton = Lr0.canonical_lr0_automaton env in
  Dump.dump_lr0_automaton env automaton;

  let (first, eps) = First_follow.compute_first def.grm in
```

```
    let follow = First_follow.compute_follow env (first, eps) in
    let tables = Slr.lr_tables env automaton follow in
    Dump.dump_lrtables env tables;

    let oc = open_out dest_name in
    Output.output_parser def env tables ic oc;
    close_out oc;
    ()
```

⟨*toplevel* `Main._1`*(yacc)* 41a⟩≡ (100a)

```
  let _ =
  (*
    Tests.test_lr0 ();
    Tests.test_first_follow ();
    Tests.test_slr ();
    Tests.test_lr_engine ();
  *)
    (*Printexc.catch*) main ();
    exit 0
```

⟨*function* `Ast.start_symbol`*(yacc)* 41b⟩≡ (91)

```
  let start_symbol def =
    try
      (match
        def.directives |> List.find (function
          | Start _x -> true
          | _ -> false
        )
       with
       | Start x -> x
       | _ -> failwith "impossible"
      )
    with Not_found -> failwith "no start symbol found"
```

Chapter 13

Lexing

⟨yacc/lexer.mll 42a⟩≡
```
{
⟨copyright ocamlyacc 8b⟩

open Ast (* Location *)
open Parser

exception Lexical_error of string

⟨Lexer helper functions and globals(yacc) 43c⟩
}

⟨rule Lexer.main(yacc) 42c⟩

⟨rule Lexer.action(yacc) 44c⟩

⟨rule Lexer.comment(yacc) 43d⟩

⟨rule Lexer.angle(yacc) 44e⟩

⟨backward compatible lexing rules(yacc) 99a⟩
```

⟨type Parser.token(yacc) 42b⟩≡ (45a)
```
%token <Ast.term> TTerm
%token <Ast.nonterm> TNonterm
%token Ttoken Tprec Tstart Ttype
%token TColon TOr TSemicolon
%token <string> TAngle
%token <Ast.location> TAction
%token TEOF
```

⟨rule Lexer.main(yacc) 42c⟩≡ (42a)
```
rule main = parse
  ⟨Lexer.main() space case (yacc) 43a⟩
  ⟨Lexer.main() comment case (yacc) 43b⟩
  ⟨Lexer.main() keyword or identifier case (yacc) 43e⟩
  ⟨Lexer.main() action case (yacc) 44a⟩
  ⟨Lexer.main() operator cases (yacc) 43f⟩
  ⟨Lexer.main() type case (yacc) 44d⟩
  ⟨Lexer.main() backward compatible cases (yacc) 99b⟩
| eof  { TEOF }
| _
    { raise(Lexical_error
              ("illegal character " ^ String.escaped(Lexing.lexeme lexbuf))) }
```

13.1 Comments

⟨Lexer.main() *space case (yacc)* 43a⟩≡ (42c)
```
  | [' ' '\010' '\013' '\009' '\012' ] +
      { main lexbuf }
```

⟨Lexer.main() *comment case (yacc)* 43b⟩≡ (42c)
```
  | "(*"
      { comment_depth := 1;
        comment lexbuf;
        main lexbuf }
```

⟨*Lexer helper functions and globals(yacc)* 43c⟩≡ (42a) 44b ▷
```
  let comment_depth = ref 0
```

⟨*rule Lexer.comment(yacc)* 43d⟩≡ (42a)
```
  and comment = parse
  | "(*"
      { incr comment_depth;
        comment lexbuf }
  | "*)"
      { decr comment_depth;
        if !comment_depth == 0
        then ()
        else comment lexbuf }

  | eof { raise(Lexical_error "unterminated comment") }
  | _ { comment lexbuf }
```

13.2 Keywords and identifiers

⟨Lexer.main() *keyword or identifier case (yacc)* 43e⟩≡ (42c)
```
  (* terminals, uppercase *)
  | ['A'-'Z' ] ['A'-'Z' 'a'-'z' '\'' '_' '0'-'9'] *
      { TTerm (T (Lexing.lexeme lexbuf)) }
  (* nonterminals, lowercase *)
  | ['a'-'z' ] ['A'-'Z' 'a'-'z' '\'' '_' '0'-'9'] *
      { TNonterm (NT (Lexing.lexeme lexbuf)) }
  (* directives, % prefixed *)
  | '%' ['A'-'Z' 'a'-'z'] ['A'-'Z' 'a'-'z' '\'' '_' '0'-'9'] *
      { match Lexing.lexeme lexbuf with
        | "%token" -> Ttoken
        | "%prec" -> Tprec
        | "%start" -> Tstart
        | "%type" -> Ttype
        | s -> failwith ("Unknown directive: " ^ s)
      }
```

13.3 Operators

⟨Lexer.main() *operator cases (yacc)* 43f⟩≡ (42c)
```
  | ':'  { TColon }
  | '|'  { TOr }
  | ';'  { TSemicolon }
```

13.4 Actions

⟨Lexer.main() *action case (yacc)* 44a⟩≡ (42c)
```
  (* actions and header/trailer *)
  | '{'
      { let n1 = Lexing.lexeme_end lexbuf in
        brace_depth := 1;
        let n2 = action lexbuf in
        TAction(Location(n1, n2)) }
```

⟨*Lexer helper functions and globals(yacc)* 44b⟩+≡ (42a) ◁43c
```
  let brace_depth = ref 0
```

⟨*rule Lexer.action(yacc)* 44c⟩≡ (42a)
```
  (* TODO: handle $x *)
  and action = parse
  | '{'
      { incr brace_depth;
        action lexbuf }
  | '}'
      { decr brace_depth;
        if !brace_depth == 0
        then Lexing.lexeme_start lexbuf
        else action lexbuf }
  | "(*"
      { comment_depth := 1;
        comment lexbuf;
        action lexbuf }

  | eof { raise (Lexical_error "unterminated action") }
  | _   { action lexbuf }
```

13.5 Types

⟨Lexer.main() *type case (yacc)* 44d⟩≡ (42c)
```
  (* for types *)
  | '<'  { TAngle (angle lexbuf) }
```

⟨*rule Lexer.angle(yacc)* 44e⟩≡ (42a)
```
  and angle = parse
  | '>' { "" }
  | eof { raise(Lexical_error "unterminated type") }
  | [^'>']+ { let s = Lexing.lexeme lexbuf in s ^ angle lexbuf }
  | _ { let s = Lexing.lexeme lexbuf in s ^ angle lexbuf }
```

Chapter 14

Parsing

14.1 Overview

⟨yacc/parser.mly 45a⟩≡
```
%{
⟨copyright ocamlyacc 8b⟩
open Ast

%}

⟨type Parser.token(yacc) 42b⟩

⟨Parser entry point definition(yacc) 45c⟩

%%

⟨grammar(yacc) 45b⟩
```

⟨grammar(yacc) 45b⟩≡ (45a)
```
⟨yacc top rule 45d⟩

⟨yacc header rule 46a⟩

⟨yacc directive rule 46h⟩

⟨yacc grammar rule 46c⟩

⟨yacc rule rule 46d⟩

⟨yacc extra rules 46b⟩
```

14.2 Parser definition entry point

⟨Parser entry point definition(yacc) 45c⟩≡ (45a)
```
%start parser_definition
%type <Ast.parser_definition> parser_definition
```

⟨yacc top rule 45d⟩≡ (45b)
```
parser_definition: header directives_opt grammar header_opt TEOF
  { { header = $1; directives = $2; grm = $3; trailer = $4 } }
;
```

14.3 Header and trailer

⟨*yacc header rule* 46a⟩≡ (45b)
```
header: TAction { $1 }
;
```

⟨*yacc extra rules* 46b⟩≡ (45b) 46g▷
```
header_opt:
          { Ast.noloc }
 | header { $1 }
;
```

14.4 Grammar rule

⟨*yacc grammar rule* 46c⟩≡ (45b)
```
grammar: rules_opt { $1 }
;
```

⟨*yacc rule rule* 46d⟩≡ (45b) 46e▷
```
rule_: TNonterm TColon cases TSemicolon
  { $3 |> List.map (fun (case, action) ->
    { lhs = $1; rhs = case; act = action })
  }
;
```

⟨*yacc rule rule* 46e⟩+≡ (45b) ◁46d 46f▷
```
cases:
   symbols_opt TAction           { [$1, $2] }
 | symbols_opt TAction TOr cases { ($1, $2)::$4 }
;
```

⟨*yacc rule rule* 46f⟩+≡ (45b) ◁46e
```
symbol:
   TTerm    { Term $1 }
 | TNonterm { Nonterm $1 }
;
```

⟨*yacc extra rules* 46g⟩+≡ (45b) ◁46b 47▷
```
rules_opt:
                    { [] }
 | rule_ rules_opt { $1 @ $2 }
;
symbols_opt:
                      { [] }
 | symbol symbols_opt { $1::$2 }
;
```

14.5 Directives

⟨*yacc directive rule* 46h⟩≡ (45b)
```
directive:
   Ttoken type_opt terms { $3 |> List.map (fun t -> Token ($2, t)) }
 | Tprec                 { [Prec ()] }
 | Tstart TNonterm       { [Start $2] }
 | Ttype TAngle TNonterm { [Type ($2, $3)] }
;
```

⟨*yacc extra rules* 47⟩+≡ (45b) ◁46g

```
directives_opt:
                              { [] }
 | directive directives_opt { $1 @ $2 }
;
type_opt:
          { None }
 | TAngle { Some $1 }
;
terms: TTerm terms_opt { $1::$2 }
;

terms_opt:
                    { [] }
 | TTerm terms_opt { $1::$2 }
;
```

Chapter 15

Checking

⟨*type* `Check.error` *(yacc)* 48a⟩≡ (92)
```
type error = unit
```

⟨*exception* `Check.Error` *(yacc)* 48b⟩≡ (92)
```
exception Error of error
```

⟨*signature* `Check.check` *(yacc)* 48c⟩≡ (92a)
```
val check: Ast.parser_definition -> unit
```

⟨*function* `Check.check` *(yacc)* 48d⟩≡ (92b)
```
let check def =
  failwith "TODO"
```

Chapter 16

Compiling

16.1 LR0 automaton

⟨*function* `Tests.test_lr0`*(yacc)* 49a⟩≡ (98)
```
let test_lr0 () =
  let env = Lr0.mk_env_augmented_grammar (NT "e") arith in

  (* closure *)
  let items = Set.singleton (R 0, D 0) in
  let i0 = Lr0.closure env items in
  let _xs = Set.elements i0 in
  (* [(R 0, D 0); (R 1, D 0); (R 2, D 0); (R 3, D 0); (R 4, D 0); (R 5, D 0);
   (R 6, D 0)] *)

  (* goto *)
  let items = Set.of_list [(R 0, D 1); (R 1, D 1)] in
  let i6 = Lr0.goto env items (Term (T "PLUS")) in
  let _xs = Set.elements i6 in
  (* [(R 1, D 2); (R 3, D 0); (R 4, D 0); (R 5, D 0); (R 6, D 0)] *)
  ()
```

16.1.1 `automaton`

⟨*type* `Lr0.automaton`*(yacc)* 49b⟩≡ (93)
```
type automaton = {
  states: items Set_.t;
  (* state 0 is the starting state *)
  int_to_state: items array;
  state_to_int: (items, stateid) Map_.t;
  (* goto mapping *)
  trans: (items * Ast.symbol, items) Map_.t;
}
```

⟨*type* `Lr0.items`*(yacc)* 49c⟩≡ (93)
```
(* a.k.a an LR0 state *)
type items = item Set_.t
```

⟨*type* `Lr0.item`*(yacc)* 49d⟩≡ (93)
```
(* as mentionned in the dragon book *)
type item = ruleidx * dotidx
```

⟨*type* `Lr0.dotidx`*(yacc)* 49e⟩≡ (93)
```
(* the dot position in the rhs of a rule *)
type dotidx = D of int
```

16.1.2 closure()

⟨*signature* Lr0.closure*(yacc)* 50a⟩≡ (93a)
```
val closure: env -> items -> items
```

⟨*function* Lr0.closure*(yacc)* 50b⟩≡ (93b)
```
(* todo: opti: use kernel items *)
let closure env items =
  let result = ref items in

  let added = ref true in
  while !added do
    added := false;

    !result |> Set.iter (fun item ->
      let (R ridx), didx = item in
      let r = env.g.(ridx) in

      match after_dot r didx with
      | Some (Nonterm b) ->
          let rules_idx = rules_of b env in
          rules_idx |> List.iter (fun ridx ->
            let item = (ridx, D 0) in
            if not (Set.mem item !result) then begin
              added := true;
              result := Set.add item !result;
            end
          )
      | _ -> ()
    )
  done;
  !result
```

⟨*signature* Lr0.after_dot*(yacc)* 50c⟩≡ (93a)
```
val after_dot: Ast.rule -> dotidx -> Ast.symbol option
```

⟨*function* Lr0.after_dot*(yacc)* 50d⟩≡ (93b)
```
let after_dot r (D idx) =
  try Some (List.nth r.rhs idx)
  with Failure _ -> None
```

⟨*function* Lr0.rules_of*(yacc)* 50e⟩≡ (93b)
```
let rules_of nt env =
  let res = ref [] in
  env.g |> Array.iteri (fun idx r ->
    if r.lhs = nt
    then res := (R idx) :: !res
  );
  List.rev !res
```

16.1.3 goto()

⟨*signature* Lr0.goto*(yacc)* 50f⟩≡ (93a)
```
val goto: env -> items -> Ast.symbol -> items
```

⟨*function* `Lr0.goto` *(yacc)* 51a⟩≡ (93b)
```
let goto env items symbol =
  let start =
    Set.fold (fun item acc ->
      let (R ridx), didx = item in
      let r = env.g.(ridx) in
      match after_dot r didx with
      | Some symbol2 when symbol = symbol2 ->
          Set.add (R ridx, move_dot_right didx) acc
      | _ -> acc
    ) items Set.empty
  in
  closure env start
```

⟨*function* `Lr0.move_dot_right` *(yacc)* 51b⟩≡ (93b)
```
let move_dot_right (D idx) =
  (D (idx + 1))
```

16.1.4 canonical_lr0_automaton()

⟨*signature* `Lr0.mk_env_augmented_grammar` *(yacc)* 51c⟩≡ (93a)
```
val mk_env_augmented_grammar: Ast.nonterm (* start *) -> Ast.grammar -> env
```

⟨*function* `Lr0.mk_env_augmented_grammar` *(yacc)* 51d⟩≡ (93b)
```
let mk_env_augmented_grammar start xs =
  let noloc = Location (0, 0) in
  let start = {lhs = Ast.start_nonterminal; rhs =[Nonterm start]; act=noloc} in
  { g = Array.of_list (start::xs) }
```

⟨*signature* `Lr0.canonical_lr0_automaton` *(yacc)* 51e⟩≡ (93a)
```
(* assumes augmented grammar *)
val canonical_lr0_automaton: env -> automaton
```

⟨*function* `Lr0.canonical_lr0_automaton` *(yacc)* 51f⟩≡ (93b)
```
let canonical_lr0_automaton env =
  let start_item = (R 0, D 0) in
  let start_items = closure env (Set.singleton start_item) in
  let result = ref (Set.singleton start_items) in
  let transitions = ref Map.empty in
  let symbols = all_symbols env in

  let added = ref true in
  while !added do
    added := false;

    !result |> Set.iter (fun items ->
      symbols |> Set.iter (fun symb ->
        let itemset = goto env items symb in
        if not (Set.is_empty itemset) && not (Map.mem (items, symb)!transitions)
        then transitions := Map.add (items, symb) itemset !transitions;

        if not (Set.is_empty itemset) && not (Set.mem itemset !result)
        then begin
          added := true;
          result := Set.add itemset !result;
        end
      )
    )
  done;
```

```
  let states = !result in
  let trans = !transitions in

  (* put start state first in the list of states *)
  let states_without_start = Set.remove start_items states |> Set.elements  in
  let states_list = start_items::states_without_start in

  let int_to_items = states_list |> Array.of_list in
  let items_to_int =
    let x = ref Map.empty in
    int_to_items |> Array.iteri (fun i items ->
      x := Map.add items (S i) !x
    );
    !x
  in
  { states = states;
    int_to_state = int_to_items;
    state_to_int = items_to_int;
    trans = trans
  }
```

⟨*signature* `Lr0.all_symbols` *(yacc)* 52a⟩≡ (93a)

```
val all_symbols: env -> Ast.symbol Set_.t
```

⟨*function* `Lr0.all_symbols` *(yacc)* 52b⟩≡ (93b)

```
let all_symbols env =
  env.g |> Array.fold_left (fun acc r ->
    ((Nonterm r.lhs)::r.rhs) |> List.fold_left (fun acc symbol ->
      Set.add symbol acc
      ) acc
  ) Set.empty
```

16.2 SLR tables

⟨*signature* `Slr.lr_tables` *(yacc)* 52c⟩≡ (95b)

```
val lr_tables:
  Lr0.env -> Lr0.automaton -> First_follow.follow -> Lrtables.lr_tables
```

⟨*function* `Tests.test_slr` *(yacc)* 52d⟩≡ (98)

```
let test_slr () =
  let env = Lr0.mk_env_augmented_grammar (NT "e") arith in

  (* automaton *)
  let auto = Lr0.canonical_lr0_automaton env in

  (* first, follow *)
  let (first, eps) = First_follow.compute_first arith in
  let follow = First_follow.compute_follow env (first, eps) in

  (* slr tables *)
  let tables = Slr.lr_tables env auto follow in

  Dump.dump_lrtables env tables;
  ()
```

16.2.1 first

⟨*constant* Tests.arith_ll*(yacc)* 53a⟩≡ (98)

```
(*
 * E -> E E'
 * E' -> + T E' | epsilon
 * T -> F T'
 * T' -> * F T' | epsilon
 * F -> ( E ) | id
 *)
let arith_ll =
    [
     {lhs = NT "e";
      rhs = [Nonterm (NT "t"); Nonterm (NT "e'")];
      act = noloc};
     {lhs = NT "e'";
      rhs = [Term (T "PLUS"); Nonterm (NT "t"); Nonterm (NT "e'")];
      act = noloc};
     {lhs = NT "e'";
      rhs = [];
      act = noloc};
     {lhs = NT "t";
      rhs = [Nonterm (NT "f"); Nonterm (NT "t'")];
      act = noloc};
     {lhs = NT "t'";
      rhs = [Term (T "MULT"); Nonterm (NT "f"); Nonterm (NT "t'")];
      act = noloc};
     {lhs = NT "t'";
      rhs = [];
      act = noloc};
     {lhs = NT "f";
      rhs = [Term (T "TOPAR"); Nonterm (NT "e"); Term (T "TCPAR")];
      act = noloc};
     {lhs = NT "f";
      rhs = [Term (T "ID")];
      act = noloc}]
```

⟨*function* Tests.test_first_follow*(yacc)* 53b⟩≡ (98)

```
let test_first_follow () =
  let (first, eps) = First_follow.compute_first arith_ll in
  let _first' = first |> Map.to_list |> List.map (fun (t, set) ->
    t, Set.elements set)
  in
  let _eps' = Set.elements eps in

  let env = Lr0.mk_env_augmented_grammar (NT "e") arith_ll in
  let follow = First_follow.compute_follow env (first, eps) in
  let _follow' = follow |> Map.to_list |> List.map (fun (t, set) ->
    t, Set.elements set)
  in
  ()
```

⟨*type* First_follow.first*(yacc)* 53c⟩≡ (94)

```
type first = (Ast.symbol, Ast.term Set_.t) Map_.t
```

⟨*type* First_follow.epsilon*(yacc)* 53d⟩≡ (94)

```
type epsilon = Ast.nonterm Set_.t
```

⟨*signature* First_follow.compute_first*(yacc)* 53e⟩≡ (94a)

```
val compute_first: Ast.grammar -> first * epsilon
```

⟨*function* `First_follow.compute_first` *(yacc)* 54⟩≡ (94b)

```
let compute_first grm =
  (* faster to use an hashtbl? could use Hashtbl.replace which would
   * be faster?
   *)
  let first = ref Map.empty in
  let epsilon = ref Set.empty in

  (* initialize first *)
  grm |> List.iter (fun r ->
    if not (Map.mem (Nonterm r.lhs) !first)
    then first := !first |> Map.add (Nonterm r.lhs) Set.empty ;
    r.rhs |> List.iter (function
      | Term t ->
        if not (Map.mem (Term t) !first)
        then first := !first |> Map.add (Term t) (Set.singleton t);
      | Nonterm _  -> ()
    )
  );

  (* fixpoint *)
  let added = ref true in
  while !added do
    added := false;

    grm |> List.iter (fun r ->
      let lhs = Nonterm r.lhs in
      if r.rhs = [] && not (Set.mem r.lhs !epsilon)
      then begin
        added := true;
        epsilon := Set.add r.lhs !epsilon;
      end;

      let rec aux all_before_are_nullable xs =
        match xs with
        | [] ->
            if all_before_are_nullable && not (Set.mem r.lhs !epsilon)
            then begin
              epsilon := Set.add r.lhs !epsilon;
              added := true;
            end
        | x::xs when all_before_are_nullable ->
            (* less: if check.ml use/def check has been done correctly,
             * we should never get some Not_found here
             *)
             let first_x = try Map.find x !first with Not_found -> Set.empty in
             let old = try Map.find lhs !first with Not_found -> Set.empty in
             if not (Set.is_empty first_x) &&
                not (Set.subset first_x old)
             then begin
               first := !first |> Map.add(Nonterm r.lhs)(Set.union old first_x);
               added := true
             end;
             (match x with
             | Term _ -> (* we can stop here *) ()
             | Nonterm nt ->
                if Set.mem nt !epsilon
                then aux true xs
             )
        | _ -> ()
```

```
      in
      aux true r.rhs
    )
  done;
  !first, !epsilon
```

16.2.2 follow

⟨*type* First_follow.follow*(yacc)* 55a⟩≡ (94)

```
type follow = (Ast.nonterm, Ast.term Set_.t) Map_.t
```

⟨*signature* First_follow.compute_follow*(yacc)* 55b⟩≡ (94a)

```
(* take Lr0.env for its augmented grammar *)
val compute_follow: Lr0.env -> first * epsilon -> follow
```

⟨*function* First_follow.compute_follow*(yacc)* 55c⟩≡ (94b)

```
(* assumes augmented grammar *)
let compute_follow env (first, epsilon) =
  let follow = ref Map.empty in

  (* initialize first *)
  env.g |> Array.iter (fun r ->
    if not (Map.mem r.lhs !follow)
    then follow := !follow |> Map.add r.lhs Set.empty ;
  );
  (* follow($S) = { $ } *)
  follow := !follow |> Map.add Ast.start_nonterminal
      (Set.singleton Ast.dollar_terminal);

  (* fixpoint *)
  let added = ref true in
  while !added do
    added := false;

    env.g |> Array.iter (fun r ->
      let rec aux xs =
        match xs with
        | [] -> ()
        | x::beta ->
          (match x with
          (* A -> alpha B beta *)
          | Nonterm b ->
              let set = first_of_sequence (first, epsilon) beta in
              let oldb = Map.find b !follow in
              if not (Set.is_empty set) &&
                 not (Set.subset set oldb) then begin
                  follow := !follow |> Map.add b (Set.union oldb set);
                  added := true
                end;
              if epsilon_of_sequence epsilon beta
              then begin
                let oldb = Map.find b !follow in
                let follow_a = Map.find r.lhs !follow in
                if not (Set.is_empty follow_a) &&
                   not (Set.subset follow_a oldb) then begin
                     follow := !follow |> Map.add b (Set.union oldb follow_a);
                     added := true
                   end;
              end;
```

```
            aux beta
        | Term _ -> aux beta
        )
    in
    aux r.rhs
  )
done;
!follow
```

⟨*function* First_follow.first_of_sequence*(yacc)* 56a⟩≡ (94b)

```
let rec first_of_sequence (first, epsilon) xs =
  match xs with
  | [] -> Set.empty
  | x::xs ->
      let set = Map.find x first in
      (match x with
      | Term _ -> set
      | Nonterm nt ->
          if Set.mem nt epsilon
          then Set.union set (first_of_sequence (first, epsilon) xs)
          else set
      )
```

⟨*function* First_follow.epsilon_of_sequence*(yacc)* 56b⟩≡ (94b)

```
let rec epsilon_of_sequence epsilon xs =
  match xs with
  | [] -> true
  | x::xs ->
      (match x with
      | Term _ -> false
      | Nonterm nt ->
          Set.mem nt epsilon && epsilon_of_sequence epsilon xs
      )
```

16.2.3 Slr.lr_tables()

⟨*function* Slr.lr_tables*(yacc)* 56c⟩≡ (96a)

```
let lr_tables env auto follow =
  let trans = auto.trans |> Map.to_list in

  let action_tables =
    Map.fold (fun items stateid acc ->
      Set.fold (fun item acc ->
        let (R ridx, didx) = item in
        let r = env.g.(ridx) in
        match Lr0.after_dot r didx with
        (* a shift *)
        | Some (Term t) ->
            let items2 = Map.find (items, (Term t)) auto.trans in
            let dst = Map.find items2 auto.state_to_int in
            ((stateid, t), Shift dst)::acc
        | Some (Nonterm _) -> acc
        (* a reduction *)
        | None ->
            if r.lhs = Ast.start_nonterminal
            then ((stateid, Ast.dollar_terminal), Accept)::acc
            else
              let terms = Map.find r.lhs follow in
              let xs = Set.elements terms in
```

```
            (xs |> List.map (fun t -> (stateid, t), Reduce (R ridx))) @ acc
    ) items acc
  ) auto.state_to_int []
in


let goto_tables =
  trans |> map_filter (fun ((items1, symb), items2) ->
    match symb with
    | Nonterm nt ->
      let src = Map.find items1 auto.state_to_int in
      let dst = Map.find items2 auto.state_to_int in
      Some ((src, nt), dst)
    | _ -> None
  )
in

action_tables, goto_tables
```

Chapter 17

Generating

⟨*signature* `Output.output_parser` *(yacc)* 58a⟩≡ (96d)

```
val output_parser:
  Ast.parser_definition -> Lr0.env -> Lrtables.lr_tables ->
  in_channel -> out_channel -> unit
```

⟨*function* `Output.output_parser` *(yacc)* 58b⟩≡ (96e)

```
let output_parser def env lrtables ic oc =
  let pf x = Printf.fprintf oc x in
  let (action_table, goto_table) = lrtables in

  let htype = Hashtbl.create 13 in
  def.directives |> List.iter (function
    | Token (Some s, t) -> Hashtbl.add htype (Term t) s
    | Type (s, nt) -> Hashtbl.add htype (Nonterm nt) s
    | _ -> ()
  );

  pf "type token =\n";
  def.directives |> List.iter (function
    | Token (sopt, (T s)) ->
      pf " | %s%s\n" s
        (match sopt with
        | None -> ""
        | Some s -> spf " of %s" s
        )
    | _ -> ()
  );

  copy_chunk ic oc def.header;
  pf "\n";
  pf "let user_actions = [|\n";
  env.g |> Array.iteri (fun i r ->
    if i = 0
    then pf "  (fun __parser_env -> failwith \"parser\");\n"
    else begin
      let s = get_chunk ic r.act in
      (* ugly: right now have to be before replace_dollar_underscore
       * because replace_dollar_underscore does side effect on s
       *)
      let dollars = extract_dollars_set s in
      let s' = replace_dollar_underscore s in
      let symbs = Array.of_list (Nonterm (NT "_fake_"):: r.rhs) in
      pf "  (fun __parser_env -> \n";
      dollars |> Set.iter (fun i ->
        pf "     let _%d = (Parsing.peek_val_simple __parser_env %d : %s) in\n"
```

```
        i i
        (* type info on terminal or on non terminal *)
        (let symb = symbs.(i) in
         if not (Hashtbl.mem htype symb)
         then
           (match symb with
           | Nonterm (NT s) -> "'" ^ s
           | Term _ -> failwith "you try to access a token with no value"
           )
         else Hashtbl.find htype symb
        )
    );

    pf "    Obj.repr((\n";
    pf "      ";
    pf "%s" s';
    pf "   )";
    if Hashtbl.mem htype (Nonterm r.lhs)
    then pf ": %s" (Hashtbl.find htype (Nonterm r.lhs))
    else ();
    pf ")\n";
    pf "   );\n";
  end
);
pf "|]\n";
copy_chunk ic oc def.trailer;


pf "\n";
pf "open Parsing\n";

(* for debugging support *)
pf "let string_of_token = function\n";
def.directives |> List.iter (function
  | Token (sopt, (T s)) ->
    pf " | %s%s -> \"%s\"\n" s
      (match sopt with
      | None -> ""
      | Some _s -> " _"
      )
      s
  | _ -> ()
);
pf "\n";

(* the main tables *)
pf "let lrtables = {\n";

(* the action table *)
pf "  action = (function\n";
action_table |> List.iter (fun ((S id, T t), action) ->
  (* if reached a state where there is dollar involved, means
   * we're ok!
   *)
  if t = "$"
  then begin
    (* in practice the '_' will be the TEOF token returned another time
     * by the lexer.
     *)
    pf "   | S %d, _ -> " id;
```

```
    (match action with
    | Reduce (R ridx) ->
        let r = env.g.(ridx) in
        let n = List.length r.rhs in
        let (NT l) = r.lhs in
        pf "Reduce (NT \"%s\", %d, RA %d)" l n ridx
    | Accept -> pf "Accept"
    | _ -> failwith "impossible to have a non reduce or accept action on $"
    );
    pf "\n";
  end
  else begin
    pf "   | S %d, %s%s -> " id t
      (if Hashtbl.mem htype (Term (T t))
       then " _"
       else ""
      );
    (match action with
    | Shift (S id) -> pf "Shift (S %d)" id
    | Accept -> pf "Accept"
    | Reduce (R ridx) ->
        let r = env.g.(ridx) in
        let n = List.length r.rhs in
        let (NT l) = r.lhs in
        pf "Reduce (NT \"%s\", %d, RA %d)" l n ridx
    | Error -> failwith "Error should not be in action tables"
    );
    pf "\n";
  end
);

pf "    | _ -> raise Parse_error\n";
pf "  );\n";


(* the goto table *)
pf "  goto = (function\n";
goto_table |> List.iter (fun ((S id1, NT nt), S id2) ->
  pf "  | S %d, NT \"%s\" -> S %d\n" id1 nt id2
);
pf "    | _ -> raise Parse_error\n";
pf "  );\n";
pf "}\n";


(* the main entry point *)
let nt = Ast.start_symbol def in
let (NT start) = nt in

pf "let %s lexfun lexbuf =\n" start;
pf "  Parsing.yyparse_simple lrtables user_actions lexfun string_of_token lexbuf\n";
()
```

17.1 Entry point

17.2 LR tables

17.3 User actions

17.3.1 Text extraction

⟨*constant* `Output.copy_buffer` *(yacc)* 61a⟩≡ (96e)
```
let copy_buffer = Bytes.create 1024
```

⟨*function* `Output.get_chunk` *(yacc)* 61b⟩≡ (96e)
```
let get_chunk ic (Location(start,stop)) =
  seek_in ic start;
  let buf = Buffer.create 1024 in

  let n = ref (stop - start) in
  while !n > 0 do
    let m = input ic copy_buffer 0 (min !n 1024) in
    Buffer.add_string buf (Bytes.sub_string copy_buffer 0 m);
    n := !n - m
  done;
  Buffer.contents buf
```

⟨*function* `Output.copy_chunk` *(yacc)* 61c⟩≡ (96e)
```
let copy_chunk ic oc (Location(start,stop)) =
  seek_in ic start;
  let n = ref (stop - start) in
  while !n > 0 do
    let m = input ic copy_buffer 0 (min !n 1024) in
    output oc copy_buffer 0 m;
    n := !n - m
  done
```

17.3.2 Dollar substitutions

⟨*function* `Output.replace_dollar_underscore` *(yacc)* 61d⟩≡ (96e)
```
(* less: we could use Str instead, but this adds a dependency.
 *  Str.global_replace (Str.regexp_string "$") "_" s
 *)
let replace_dollar_underscore s =
  let buf = Bytes.of_string s in
  let rec aux startpos =
    try
      let idx = Bytes.index_from buf startpos '$' in
      Bytes.set buf idx '_';
      aux (idx+1)
    with Not_found -> ()
  in
  aux 0;
  Bytes.to_string buf
```

17.3.3 Semantic values managment

⟨*function* `Output.int_of_char` *(yacc)* 62a⟩≡ (96e)

```
(* todo: what about $22? what error message give if type and $x ? *)
let int_of_char c =
  let i = Char.code c in
  if i <= Char.code '9' && i >= Char.code '1'
  then i - Char.code '0'
  else failwith (spf "the characted %c is not a char" c)
```

⟨*function* `Output.extract_dollars_set` *(yacc)* 62b⟩≡ (96e)

```
let extract_dollars_set s =
  let set = ref Set.empty in
  let rec aux startpos =
    try
      let idx = String.index_from s startpos '$' in
      let c = String.get s (idx + 1) in
      set := Set.add (int_of_char c) !set;
      aux (idx + 2)
    with Not_found | Invalid_argument _ | Failure _ -> ()
  in
  aux 0;
  !set
```

Chapter 18

Running

⟨*function* `Parsing.yyparse_simple`*(yacc)* 63⟩≡ (106)

```
let yyparse_simple lrtables rules_actions lexfun string_of_tok lexbuf =

  let env = {
    states = Stack.create ();
    values = Stack.create ();
    current_rule_len = 0;
  }
  in

  env.states |> Stack.push (S 0);
  let a = ref (lexfun lexbuf) in

  let finished = ref false in
  let res = ref (Obj.repr ()) in
  while not !finished do

    let s = Stack.top env.states in
    log (spf "state %d, tok = %s" (let (S x) = s in x) (string_of_tok !a));

    match lrtables.action (s, !a) with
    | Shift t ->
        log (spf "shift to %d" (let (S x) = t in x));
        env.states |> Stack.push t;
        env.values |> Stack.push (value_of_tok (Obj.repr !a));
        a := lexfun lexbuf;
    | Reduce (nt, n, ra) ->
        for i = 1 to n do
          Stack.pop env.states |> ignore
        done;
        let s = Stack.top env.states in
        env.states |> Stack.push (lrtables.goto (s, nt));
        let (NT ntstr) = nt in
        let (RA raidx) = ra in
        env.current_rule_len <- n;
        let v = rules_actions.(raidx) env in
        for i = 1 to n do
          Stack.pop env.values |> ignore
        done;
        env.values |> Stack.push v;
        log (spf "reduce %s, ra = %d" ntstr raidx);
    | Accept ->
        log "done!";
        finished := true;
```

```
        res := Stack.top env.values;
    done;
    Obj.magic !res
```

⟨*signature* `Parsing.peek_val_simple`*(yacc)* 64a⟩≡ (101h)
```
  val peek_val_simple: parser_env_simple -> int -> 'a
```

⟨*function* `Parsing.peek_val_simple`*(yacc)* 64b⟩≡ (106)
```
  let peek_val_simple env i =
    if i < 1 && i >= env.current_rule_len
    then failwith (spf "peek_val_simple invalid argument %d" i)
    else Obj.magic (Common.Stack_.nth (env.current_rule_len - i) env.values)
```

⟨*function* `Parsing.value_of_tok`*(yacc)* 64c⟩≡ (106)
```
  (* hmm, imitate what is done in parsing.c. A big ugly but tricky
   * to do otherwise and have a generic LR parsing engine.
   *)
  let value_of_tok t =
    if Obj.is_block t
    then Obj.field t 0
    else Obj.repr ()
```

Part III

Advanced Topics

Chapter 19

Advanced Features

19.1 `as`

19.2 Characters as terminals

19.3 `left`, `right`, `assoc`

19.4 Action in the middle of a rule

Chapter 20

Error managment Support

20.1 Positions tracking

20.2 Error recovery

Chapter 21

Debugging Support

21.1 #line

21.2 Traces

⟨*constant* `Parsing.debug`*(yacc)* 68a⟩≡ (106)

```
let debug = ref true
```

⟨*function* `Parsing.log`*(yacc)* 68b⟩≡ (106)

```
let log x =
  if !debug
  then begin
    print_endline ("YACC: " ^ x); flush stdout
  end
```

21.3 parser.output

Chapter 22

Optimisations

22.1 Lex

22.1.1 Compacted automata

⟨*type* Compact.lex_tables 69a⟩≡ (86)

```
(* Compaction of an automata *)

type lex_tables =
  {
    (* negative int -(n+1)  for Perform n, idx in tbl_trans for Shift *)
    tbl_base: int array;                (* Perform / Shift *)
    (* -1 = No_remember, positive = Remember n *)
    tbl_backtrk: int array;             (* No_remember / Remember *)

    tbl_default: int array;             (* Default transition *)
    tbl_trans: int array;               (* Transitions (compacted) *)
    tbl_check: int array;               (* Check (compacted) *)
  }
```

22.1.2 compact_tables()

⟨*signature* Compact.compact_tables 69b⟩≡ (86a)

```
val compact_tables: Lexgen.automata_matrix -> lex_tables
```

⟨*function* Compact.compact_tables 69c⟩≡ (86b)

```
let compact_tables state_v =
  let n = Array.length state_v in

  let base = Array.make n 0 in
  let backtrk = Array.make n (-1) in
  let default = Array.make n 0 in


  for i = 0 to n - 1 do
    match state_v.(i) with
      Perform n ->
        base.(i) <- -(n+1)
    | Shift(trans, move) ->
        (match trans with
          No_remember -> ()
        | Remember n -> backtrk.(i) <- n
        );
        let (b, d) = pack_moves i move in
```

```
        base.(i) <- b;
        default.(i) <- d
    done;
    { tbl_base = base;
      tbl_backtrk = backtrk;
      tbl_default = default;
      tbl_trans = Array.sub !trans 0 !last_used;
      tbl_check = Array.sub !check 0 !last_used;
    }
```

⟨*global* Compact.trans 70a⟩≡ (86b)
```
  let trans = ref(Array.make 1024 0)
```

⟨*global* Compact.check 70b⟩≡ (86b)
```
  let check = ref(Array.make 1024 (-1))
```

⟨*global* Compact.last_used 70c⟩≡ (86b)
```
  let last_used = ref 0
```

⟨*function* Compact.pack_moves 70d⟩≡ (86b)
```
  let pack_moves state_num move_t =
    let move_v = Array.make 257 0 in
    for i = 0 to 256 do
      move_v.(i) <-
        (match move_t.(i) with
          Backtrack -> -1
        | Goto n -> n)
    done;

    let default = most_frequent_elt move_v in
    let nondef = non_default_elements default move_v in

    let rec pack_from b =
      while b + 257 > Array.length !trans do
        grow_transitions()
      done;
      let rec try_pack = function
        [] -> b
      | (pos, _v) :: rem ->
          if !check.(b + pos) = -1 then try_pack rem else pack_from (b+1) in
      try_pack nondef
    in
    let base = pack_from 0 in

    nondef |> List.iter (fun (pos, v) ->
        !trans.(base + pos) <- v;
        !check.(base + pos) <- state_num
    );
    if base + 257 > !last_used
    then last_used := base + 257;
    (base, default)
```

⟨*function* Compact.most_frequent_elt 70e⟩≡ (86b)
```
  (* Determine the integer occurring most frequently in an array *)

  let most_frequent_elt v =
    let frequencies = Hashtbl.create 17 in
    let max_freq = ref 0 in
    let most_freq = ref (v.(0)) in
    for i = 0 to Array.length v - 1 do
```

```
      let e = v.(i) in
      let r =
        try
          Hashtbl.find frequencies e
        with Not_found ->
          let r = ref 1 in Hashtbl.add frequencies e r; r
      in
      incr r;
      if !r > !max_freq then begin
        max_freq := !r;
        most_freq := e
      end
    done;
    !most_freq
```

⟨*function* Compact.non_default_elements 71a⟩≡ (86b)

```
  (* Transform an array into a list of (position, non-default element) *)

  let non_default_elements def v =
    let rec nondef i =
      if i >= Array.length v
      then []
      else begin
        let e = v.(i) in
        if e = def then nondef(i+1) else (i, e) :: nondef(i+1)
      end in
    nondef 0
```

⟨*function* Compact.grow_transitions 71b⟩≡ (86b)

```
  let grow_transitions () =
    let old_trans = !trans
    and old_check = !check in
    let n = Array.length old_trans in
    trans := Array.make (2*n) 0;
    Array.blit old_trans 0 !trans 0 !last_used;
    check := Array.make (2*n) (-1);
    Array.blit old_check 0 !check 0 !last_used
```

22.1.3 Output compacted tables

⟨*type* Lexing.lex_tables 71c⟩≡ (90d 88b)

```
  (* The following definitions are used by the generated scanners only.
     They are not intended to be used by user programs. *)

  type lex_tables =
    { lex_base: string;
      lex_backtrk: string;
      lex_default: string;
      lex_trans: string;
      lex_check: string }
```

22.1.4 C transition engine

22.2 Yacc

22.2.1 C transition engine

Chapter 23

Advanced Topics

23.1 Unicode

Chapter 24

Conclusion

Appendix A

Debugging

A.1 Dumpers

⟨*signature* Dump.dump_lr0_automaton*(yacc)* 74a⟩≡ (97a)
```
val dump_lr0_automaton: Lr0.env -> Lr0.automaton -> unit
```

⟨*signature* Dump.dump_item*(yacc)* 74b⟩≡ (97a)
```
val dump_item: Lr0.env -> Lr0.item -> unit
```

⟨*signature* Dump.dump_items*(yacc)* 74c⟩≡ (97a)
```
val dump_items: Lr0.env -> Lr0.items -> unit
```

⟨*signature* Dump.dump_lrtables*(yacc)* 74d⟩≡ (97a)
```
val dump_lrtables: Lr0.env -> Lrtables.lr_tables -> unit
```

⟨*function* Dump.string_of_action*(yacc)* 74e⟩≡ (97b)
```
let string_of_action x =
  match x with
  | Lrtables.Shift (S d) -> spf "s%d" d
  | Lrtables.Reduce (R d) -> spf "r%d" d
  | Lrtables.Accept -> spf "acc"
  | Lrtables.Error -> ""
```

⟨*function* Dump.dump_symbol*(yacc)* 74f⟩≡ (97b)
```
let dump_symbol s =
  print_string (string_of_symbol s)
```

⟨*function* Dump.string_of_symbol*(yacc)* 74g⟩≡ (97b)
```
let string_of_symbol s =
  match s with
  (* in ocamlyacc terminals are constructors and so are uppercase and
   * so non terminals are lowercase, but it's the opposite convention
   * used in the dragon book, so here we dump in the dragon book
   * way so it's easier to compare what we generate with what
   * the dragon book says we should generate
   *)
  | Nonterm (NT s) -> String.uppercase_ascii s
  | Term (T s) ->
    (match s with
    (* special cases for arith.mly and tests.ml grammar toy examples *)
    | "PLUS" -> "+"
    | "MULT" -> "*"
    | "TOPAR" -> "("
    | "TCPAR" -> ")"

    | _ -> String.lowercase_ascii s
    )
```

⟨*function* Dump.dump_item*(yacc)* 75a⟩≡ (97b)

```
let dump_item env item =
  let (R idx, D didx) = item in
  let r = env.g.(idx) in

  dump_symbol (Nonterm r.lhs);
  print_space ();
  print_string "->";
  open_box 0;
  print_space ();
  r.rhs |> Array.of_list |> Array.iteri (fun i s ->
    if i = didx
    then begin
      print_string ".";
      print_space ();
    end;
    dump_symbol s;
    print_space ();
  );
  if didx = List.length r.rhs then begin
      print_string ".";
  end;

  close_box ();
(*  print_space (); print_string "(R"; print_int idx; print_string ")" *)
  ()
```

⟨*function* Dump.dump_items*(yacc)* 75b⟩≡ (97b)

```
let dump_items env items =
  items
  |> Set.elements |> List.sort (fun (R a, _) (R b, _) -> a - b)
  |> List.iter (fun item ->
    open_box 0;
    dump_item env item;
    close_box ();
    print_newline ();
  )
```

⟨*function* Dump.dump_lr0_automaton*(yacc)* 75c⟩≡ (97b)

```
let dump_lr0_automaton env auto =

  open_box 0;

  (* the states *)
  auto.int_to_state |> Array.iteri (fun i items ->
    print_string "I"; print_int i; print_newline ();
    open_box 2;
    dump_items env items;
    close_box ();
    print_newline ();
  );

  (* the transitions *)
  auto.trans |> Map.iter (fun (items1, symb) items2 ->
    let (S src) = Map.find items1 auto.state_to_int in
    let (S dst) = Map.find items2 auto.state_to_int in
    print_string "I"; print_int src;
    print_string " --"; dump_symbol symb; print_string "-->";
    print_string " I"; print_int dst;
    print_newline ()
```

```
  );

  close_box ()
```

⟨*function* Dump.dump_lrtables*(yacc)* 76⟩≡ (97b)

```
let dump_lrtables env lrtables =
  let symbols = Lr0.all_symbols env in
  let (action_table, goto_table) = lrtables in
  let haction = hash_of_list action_table in
  let hgoto = hash_of_list goto_table in

  let (terms, nonterms) =
    symbols |> Set.elements |> partition_either (function
      | Term t -> Left t
      | Nonterm nt -> Right nt
    )
  in
  let terms = terms @ [Ast.dollar_terminal] in
  let max_state =
    action_table |> List.fold_left (fun acc (((S stateid), _), _) ->
      max acc stateid
    ) 0
  in

  (* print headers *)
  pf "   ";
  terms |> List.iter (fun t ->
    let s = string_of_symbol (Term t) in
    pf "%3s " s;
  );
  pf "  ";
  nonterms |> List.iter (fun nt ->
    let s = string_of_symbol (Nonterm nt) in
    pf "%3s " s;
  );
  pf "\n";

  let conflicts = ref [] in

  for i = 0 to max_state do
    pf "%2d " i;
    terms |> List.iter (fun t ->
      let xs = Hashtbl.find_all haction (S i, t) in
      (match xs with
      | [] -> pf "%3s " " "
      | [x] -> pf "%3s " (string_of_action x)
      | x::xs ->
        pf "%3s " "!?!";
        conflicts := (S i, t, x::xs)::!conflicts
      );
    );
    pf "  ";
    nonterms |> List.iter (fun nt ->
      let xs = Hashtbl.find_all hgoto (S i, nt) in
      (match xs with
      | [] -> pf "%3s " " "
      | [S d] -> pf "%3d " d
      | _x::_xs -> pf "%3s " "!?!"
      );
    );
```

```
  pf "\n";
done;
pf "\n";
pf "%d conflicts\n" (List.length !conflicts);
pf "\n";
()
```

Appendix B

Error Management

⟨`Main.main()` *report error exn* 78a⟩≡ (13b)

```
(match exn with
  Parsing.Parse_error ->
    prerr_string "Syntax error around char ";
    prerr_int (Lexing.lexeme_start lexbuf);
    prerr_endline "."
| Lexer.Lexical_error s ->
    prerr_string "Lexical error around char ";
    prerr_int (Lexing.lexeme_start lexbuf);
    prerr_string ": ";
    prerr_string s;
    prerr_endline "."
| _ -> raise exn
);
```

⟨*function* `Check.report_error` *(yacc)* 78b⟩≡ (92b)

```
let report_error err =
  failwith "TODO"
```

Appendix C

Utilities

⟨*function* `Slr.filter_some` *(yacc)* 79a⟩≡ (96a)
```
(* from my common.ml *)
let rec filter_some = function
  | [] -> []
  | None :: l -> filter_some l
  | Some e :: l -> e :: filter_some l
```

⟨*function* `Slr.map_filter` *(yacc)* 79b⟩≡ (96a)
```
let map_filter f xs = xs |> List.map f |> filter_some
```

⟨*type* `Dump.either` *(yacc)* 79c⟩≡ (97b)
```
type ('a,'b) either = Left of 'a | Right of 'b
```

⟨*function* `Dump.partition_either` *(yacc)* 79d⟩≡ (97b)
```
let partition_either f l =
  let rec part_either left right = function
  | [] -> (List.rev left, List.rev right)
  | x :: l ->
      (match f x with
      | Left  e -> part_either (e :: left) right l
      | Right e -> part_either left (e :: right) l) in
  part_either [] [] l
```

⟨*function* `Dump.hash_of_list` *(yacc)* 79e⟩≡ (97b)
```
let hash_of_list xs =
  let h = Hashtbl.create 101 in
  xs |> List.iter (fun (k, v) -> Hashtbl.replace h k v);
  h
```

⟨*constant* `Output.spf` *(yacc)* 79f⟩≡ (96e)
```
let spf = Printf.sprintf
```

⟨*constant* `Dump.spf` *(yacc)* 79g⟩≡ (97b)
```
let spf = Printf.sprintf
```

⟨*constant* `Parsing.spf` *(yacc)* 79h⟩≡ (106)
```
let spf = Printf.sprintf
```

⟨*constant* `Dump.pf` *(yacc)* 79i⟩≡ (97b)
```
let pf = Printf.printf
```

Appendix D

Extra Code

D.1 ocamllex/

D.1.1 lex/ast.ml

⟨lex/ast.ml 80a⟩≡
```
⟨copyright ocamllex 8a⟩
open Stdcompat (* for |> *)
(* The shallow abstract syntax *)

⟨type Syntax.charpos 11e⟩

⟨type Syntax.location 11d⟩

⟨type Syntax.char_ 11b⟩

let char_class c1 c2 =
  let rec cl n =
    if n > c2 then [] else n :: cl(succ n)
  in cl c1

(* CONFIG *)
let charset_size = 256
let char_eof = 256
let all_chars = char_class 0 255
(* alt:
let charset_size = 255
let char_eof = 0
let all_chars = char_class 1 255
*)

⟨type Syntax.regular_expression 11a⟩

⟨type Syntax.action 11g⟩

⟨type Syntax.rule 11f⟩

⟨type Syntax.lexer_definition 11c⟩
```

D.1.2 lex/output.mli

⟨lex/output.mli 80b⟩≡
```
⟨copyright ocamllex 8a⟩
⟨signature Output.output_lexdef 14⟩
```

```
val output_lexdef_simple:
      in_channel -> out_channel ->
      Ast.location (* header *) ->
      Lexgen.automata_entry list * Lexgen.automata_matrix ->
      Ast.location (* trailer *) ->
      unit
```

D.1.3 lex/output.ml

⟨lex/output.ml 81⟩≡

```
⟨copyright ocamllex 8a⟩
(* Output the DFA tables and its entry points *)

open Stdcompat (* for |> *)
open Printf
open Ast
open Lexgen
open Compact

⟨constant Output.copy_buffer 32⟩

⟨function Output.copy_chunk 31d⟩

⟨function Output.output_byte 31c⟩

⟨function Output.output_array 31b⟩

⟨function Output.output_tables 31a⟩

⟨function Output.output_entry 30d⟩

⟨function Output.output_lexdef 30a⟩

(******************************************************************************)
(* Simpler version *)
(******************************************************************************)

let debug = ref true

(* 1- Generating the actions *)

let output_action ic oc (i,act) =
  output_string oc ("action_" ^ string_of_int i ^ " lexbuf = (\n");
  if !debug
  then output_string oc (" log \"action_" ^ string_of_int i ^ "\";\n");
  copy_chunk ic oc act;
  output_string oc ")\nand ";
  ()

(* 2- Generating the states *)

let states = ref ([||] : Lexgen.automata_matrix)

let enumerate_vect v =
  let rec enum env pos =
    if pos >= Array.length v
```

```
    then env
    else
      try
        let pl = List.assoc v.(pos) env in
          pl := pos :: !pl; enum env (succ pos)
        with Not_found ->
          enum ((v.(pos), ref [pos]) :: env) (succ pos)
  in
  List.sort
    (fun (_e1, pl1) (_e2, pl2) -> compare (List.length !pl1)  (List.length !pl2))
    (enum [] 0)

let output_move oc = function
    Backtrack ->
      if !debug
      then output_string oc "log \"backtrack\"; ";
      output_string oc "backtrack lexbuf"
  | Goto dest ->
      match !states.(dest) with
        Perform act_num ->
          output_string oc ("action_" ^ string_of_int act_num ^ " lexbuf")
      | _ ->
          (* Many states are just Perform so this explains why there is some
           * big jumps in the generated files from e.g. state_3 to state_9
           * without any intermediate state_4 function; it's because state 4
           * was a Perform.
           *)
          output_string oc ("state_" ^ string_of_int dest ^ " lexbuf")

let output_char_for_read oc = function
    '\''  -> output_string oc "\\'"
  | '\\' -> output_string oc "\\\\"
  | '\n' -> output_string oc "\\n"
  | '\t' -> output_string oc "\\t"
  | c ->
      let n = Char.code c in
      if n >= 32 && n < 127 then
        output_char oc c
      else begin
        output_char oc '\\';
        output_char oc (Char.chr (48 + n / 100));
        output_char oc (Char.chr (48 + (n / 10) mod 10));
        output_char oc (Char.chr (48 + n mod 10))
      end

let rec output_chars oc = function
    [] ->
      failwith "output_chars"
  | [c] ->
      if c <= 255 then begin
      output_string oc "'";
      output_char_for_read oc (Char.chr c);
      output_string oc "'"
      end else output_string oc "'\\000'"
  | c::cl ->
      if c <= 255 then begin
      output_string oc "'";
      output_char_for_read oc (Char.chr c);
      output_string oc "'|";
      output_chars oc cl
```

```
      end else output_string oc "'\\000'"

let output_one_trans oc (dest, chars) =
  output_chars oc !chars;
  output_string oc " -> ";
  output_move oc dest;
  output_string oc "\n |  ";
  ()

let output_all_trans oc trans =
  output_string oc "  let c = Lexing.get_next_char lexbuf in\n";
  if !debug
  then output_string oc "  log (\"consuming:\" ^ (Char.escaped c));\n";
  output_string oc "  match c with\n    ";
  match enumerate_vect trans with
    [] ->
      failwith "output_all_trans"
  | (default, _) :: rest ->
      List.iter (output_one_trans oc) rest;
      output_string oc "_ -> ";
      output_move oc default;
      output_string oc "\nand ";
      ()

let output_state oc state_num = function
    Perform _i ->
      ()
  | Shift(what_to_do, moves) ->
      output_string oc
        ("state_"  ^ string_of_int state_num ^ " lexbuf =\n");
      if !debug
      then output_string oc("  log \"state_" ^string_of_int state_num^ "\";\n");

    (match what_to_do with
      No_remember -> ()
    | Remember i ->
        output_string oc "  lexbuf.lex_last_pos <- lexbuf.lex_curr_pos;\n";
        output_string oc ("  lexbuf.lex_last_action_simple <- Obj.magic action_" ^
                          string_of_int i ^ ";\n")
    );
    output_all_trans oc moves

(* 3- Generating the entry points *)

let rec output_entries oc = function
    [] -> failwith "output_entries"
  | entry :: rest ->
      let name = entry.auto_name in
      let state_num = entry.auto_initial_state in
      output_string oc (name ^ " lexbuf =\n");
      if !debug
      then output_string oc ("log \"" ^ name ^ "\";\n");
      output_string oc "  Lexing.start_lexing lexbuf;\n";
      output_string oc ("  state_" ^ string_of_int state_num ^ " lexbuf\n");
      match rest with
        [] -> output_string oc "\n"; ()
      | _  -> output_string oc "\nand "; output_entries oc rest

(* All together *)
```

```
let output_lexdef_simple ic oc header (initial_st, st) trailer =
  print_int (Array.length st); print_string " states, ";
(*  print_int (List.length actions); print_string " actions."; *)
  print_newline();
  (* for the labels *)
  output_string oc "open Lexing\n\n";
  if !debug
  then output_string oc
    "let log x = print_endline (\"LEX: \" ^ x); flush stdout\n";
  copy_chunk ic oc header;
  output_string oc "\nlet rec ";
  states := st;
  initial_st |> List.iter (fun entry ->
    let actions = entry.auto_actions in
    List.iter (output_action ic oc) actions;
    output_string oc "\n";
  );
  output_string oc "\n";
  for i = 0 to Array.length st - 1 do
    output_state oc i st.(i)
  done;
  output_string oc "\n";
  output_entries oc initial_st;
  output_string oc "\n";
  copy_chunk ic oc trailer
```

D.1.4 lex/lexgen.mli

⟨lex/lexgen.mli 84a⟩≡

```
⟨copyright ocamllex 8a⟩
(* Representation of automata *)

⟨type Lexgen.action_id 12b⟩

⟨type Lexgen.automata 12d⟩
⟨type Lexgen.automata_trans 12e⟩
⟨type Lexgen.automata_move 12f⟩

⟨type Lexgen.automata_entry 12a⟩

⟨type Lexgen.automata_matrix 12c⟩

⟨signature Lexgen.make_dfa 13c⟩
```

D.1.5 lex/lexgen.ml

⟨lex/lexgen.ml 84b⟩≡

```
⟨copyright ocamllex 8a⟩
(* Compiling a lexer definition *)

open Stdcompat (* for |> *)
open Ast
module Set = Set_
module Map = Map_

⟨type Lexgen.action_id 12b⟩
```

⟨*type* `Lexgen.charset_id` 25d⟩

⟨*type* `Lexgen.regexp` 25b⟩

⟨*type* `Lexgen.lexer_entry` 25c⟩

```
(* Representation of automata *)
```

⟨*type* `Lexgen.automata` 12d⟩
⟨*type* `Lexgen.automata_trans` 12e⟩
⟨*type* `Lexgen.automata_move` 12f⟩

⟨*type* `Lexgen.automata_entry` 12a⟩

⟨*type* `Lexgen.automata_matrix` 12c⟩

```
(* From shallow to deep syntax *)
```

⟨*constant* `Lexgen.chars` 25e⟩
⟨*constant* `Lexgen.chars_count` 25f⟩
⟨*constant* `Lexgen.actions` 25g⟩
⟨*constant* `Lexgen.actions_count` 25h⟩

⟨*function* `Lexgen.encode_regexp` 26c⟩

⟨*function* `Lexgen.encode_casedef` 26b⟩

⟨*function* `Lexgen.encode_lexdef` 26a⟩

```
(* To generate directly a NFA from a regular expression.
   Confer Aho-Sethi-Ullman, dragon book, chap. 3 *)
```

⟨*type* `Lexgen.transition` 26d⟩

⟨*type* `Lexgen.state` 27a⟩

⟨*function* `Lexgen.nullable` 28a⟩

⟨*function* `Lexgen.firstpos` 27g⟩

⟨*function* `Lexgen.lastpos` 29d⟩

⟨*function* `Lexgen.followpos` 29c⟩

⟨*constant* `Lexgen.no_action` 28f⟩

⟨*function* `Lexgen.split_trans_set` 28e⟩

⟨*constant* `Lexgen.state_map` 27c⟩
⟨*constant* `Lexgen.todo` 27d⟩
⟨*constant* `Lexgen.next_state_num` 27e⟩

⟨*function* `Lexgen.reset_state_mem` 27b⟩

⟨*function* `Lexgen.get_state` 28b⟩

⟨*function* `Lexgen.map_on_all_states` 28c⟩

⟨*function* `Lexgen.goto_state` 29b⟩

⟨*function* `Lexgen.transition_from` 29a⟩

⟨*function* `Lexgen.translate_state` 28d⟩

⟨*function* `Lexgen.encode_lexentries` 27f⟩

⟨*function* `Lexgen.make_dfa` 25a⟩

D.1.6 lex/compact.mli

⟨`lex/compact.mli` 86a⟩≡
⟨*copyright ocamllex* 8a⟩
⟨*type* `Compact.lex_tables` 69a⟩

⟨*signature* `Compact.compact_tables` 69b⟩

D.1.7 lex/compact.ml

⟨`lex/compact.ml` 86b⟩≡
⟨*copyright ocamllex* 8a⟩
```
(* Compaction of an automata *)

open Stdcompat (* for |> *)
open Lexgen
```

⟨*function* `Compact.most_frequent_elt` 70e⟩

⟨*function* `Compact.non_default_elements` 71a⟩

```
(* Compact the transition and check arrays *)
```

⟨*global* `Compact.trans` 70a⟩
⟨*global* `Compact.check` 70b⟩
⟨*global* `Compact.last_used` 70c⟩

⟨*function* `Compact.grow_transitions` 71b⟩

⟨*function* `Compact.pack_moves` 70d⟩

⟨*type* `Compact.lex_tables` 69a⟩

⟨*function* `Compact.compact_tables` 69c⟩

D.1.8 lex/main.ml

⟨`lex/main.ml` 86c⟩≡
⟨*copyright ocamllex* 8a⟩
```
(* The lexer generator. Command-line parsing. *)

open Stdcompat (* for |> *)
open Ast
```

⟨*function* `Main.main` 13b⟩

⟨*toplevel* `Main._1` 13a⟩

D.1.9 lex/lexer.mll

D.1.10 lex/parser.mly

⟨*copyright ocamllex bis* 87a⟩≡ (20a)

```
/**************************************************************************/
/*                                                                        */
/*                          Objective Caml                                */
/*                                                                        */
/*           Xavier Leroy, projet Cristal, INRIA Rocquencourt             */
/*                                                                        */
/*  Copyright 1996 Institut National de Recherche en Informatique et      */
/*  Automatique.  Distributed only by permission.                         */
/*                                                                        */
/**************************************************************************/
```

D.1.11 stdlib/lexing_.mli

⟨*signature* `Lexing.from_channel` 87b⟩≡ (88b)

```
val from_channel : in_channel -> lexbuf
        (* Create a lexer buffer on the given input channel.
           [Lexing.from_channel inchan] returns a lexer buffer which reads
           from the input channel [inchan], at the current reading position. *)
```

⟨*signature* `Lexing.from_string` 87c⟩≡ (88b)

```
val from_string : string -> lexbuf
        (* Create a lexer buffer which reads from
           the given string. Reading starts from the first character in
           the string. An end-of-input condition is generated when the
           end of the string is reached. *)
```

⟨*signature* `Lexing.from_function` 87d⟩≡ (88b)

```
val from_function : (bytes -> int -> int) -> lexbuf
        (* Create a lexer buffer with the given function as its reading method.
           When the scanner needs more characters, it will call the given
           function, giving it a character string [s] and a character
           count [n]. The function should put [n] characters or less in [s],
           starting at character number 0, and return the number of characters
           provided. A return value of 0 means end of input. *)
```

⟨*signature* `Lexing.lexeme` 87e⟩≡ (88b)

```
val lexeme : lexbuf -> string
        (* [Lexing.lexeme lexbuf] returns the string matched by
           the regular expression. *)
```

⟨*signature* `Lexing.lexeme_char` 87f⟩≡ (88b)

```
val lexeme_char : lexbuf -> int -> char
        (* [Lexing.lexeme_char lexbuf i] returns character number [i] in
           the matched string. *)
```

⟨*signature* `Lexing.lexeme_start` 87g⟩≡ (88b)

```
val lexeme_start : lexbuf -> int
        (* [Lexing.lexeme_start lexbuf] returns the position in the
           input stream of the first character of the matched string.
           The first character of the stream has position 0. *)
```

⟨*signature* Lexing.lexeme_end 88a⟩≡ (88b)
```
val lexeme_end : lexbuf -> int
        (* [Lexing.lexeme_end lexbuf] returns the position in the input stream
           of the character following the last character of the matched
           string. The first character of the stream has position 0. *)
```

⟨stdlib/lexing_.mli 88b⟩≡
```
⟨copyright ocamllex 8a⟩
(* Module [Lexing]: the run-time library for lexers generated by [ocamllex] *)
open Stdcompat (* for bytes *)

(*** Lexer buffers *)

⟨type Lexing.lexbuf 12g⟩

⟨signature Lexing.from_channel 87b⟩
⟨signature Lexing.from_string 87c⟩
⟨signature Lexing.from_function 87d⟩

(*** Functions for lexer semantic actions *)

        (* The following functions can be called from the semantic actions
           of lexer definitions (the ML code enclosed in braces that
           computes the value returned by lexing functions). They give
           access to the character string matched by the regular expression
           associated with the semantic action. These functions must be
           applied to the argument [lexbuf], which, in the code generated by
           [camllex], is bound to the lexer buffer passed to the parsing
           function. *)

⟨signature Lexing.lexeme 87e⟩
⟨signature Lexing.lexeme_char 87f⟩
⟨signature Lexing.lexeme_start 87g⟩
⟨signature Lexing.lexeme_end 88a⟩

(*--*)

⟨type Lexing.lex_tables 71c⟩

(* take an integer representing a state and return an integer representing
 * an action_id
 *)
external engine: lex_tables -> int -> lexbuf -> int = "lex_engine"


(* functions used by the generated scanners using the simple code generation
 *  method *)
val get_next_char : lexbuf -> char
val backtrack : lexbuf -> 'a

val start_lexing : lexbuf -> unit
```

D.1.12 stdlib/lexing_.ml

⟨*function* Lexing.lex_refill 88c⟩≡ (90d)
```
let lex_refill read_fun aux_buffer lexbuf =
  let read =
    read_fun aux_buffer (Bytes.length aux_buffer) in
```

```
    let n =
      if read > 0
      then read
      else (lexbuf.lex_eof_reached <- true; 0) in
    if lexbuf.lex_start_pos < n then begin
      let oldlen = lexbuf.lex_buffer_len in
      let newlen = oldlen * 2 in
      let newbuf = Bytes.create newlen in
      Bytes.unsafe_blit lexbuf.lex_buffer 0 newbuf oldlen oldlen;
      lexbuf.lex_buffer <- newbuf;
      lexbuf.lex_buffer_len <- newlen;
      lexbuf.lex_abs_pos <- lexbuf.lex_abs_pos - oldlen;
      lexbuf.lex_curr_pos <- lexbuf.lex_curr_pos + oldlen;
      lexbuf.lex_start_pos <- lexbuf.lex_start_pos + oldlen;
      lexbuf.lex_last_pos <- lexbuf.lex_last_pos + oldlen
    end;
    Bytes.unsafe_blit lexbuf.lex_buffer n
                      lexbuf.lex_buffer 0
                      (lexbuf.lex_buffer_len - n);
    Bytes.unsafe_blit aux_buffer 0
                      lexbuf.lex_buffer (lexbuf.lex_buffer_len - n)
                      n;
    lexbuf.lex_abs_pos <- lexbuf.lex_abs_pos + n;
    lexbuf.lex_curr_pos <- lexbuf.lex_curr_pos - n;
    lexbuf.lex_start_pos <- lexbuf.lex_start_pos - n;
    lexbuf.lex_last_pos <- lexbuf.lex_last_pos - n
```

⟨*function* Lexing.from_function 89a⟩≡ (90d)

```
  let from_function f =
    { refill_buff = lex_refill f (Bytes.create 512);
      lex_buffer = Bytes.create 1024;
      lex_buffer_len = 1024;
      lex_abs_pos = - 1024;
      lex_start_pos = 1024;
      lex_curr_pos = 1024;
      lex_last_pos = 1024;
      lex_last_action = 0;
      lex_last_action_simple = dummy_action;
      lex_eof_reached = false }
```

⟨*function* Lexing.from_channel 89b⟩≡ (90d)

```
  let from_channel ic =
    from_function (fun buf n -> input ic buf 0 n)
```

⟨*function* Lexing.from_string 89c⟩≡ (90d)

```
  let from_string s =
    { refill_buff = (fun lexbuf -> lexbuf.lex_eof_reached <- true);
      lex_buffer = Bytes.of_string s;
      lex_buffer_len = String.length s;
      lex_abs_pos = 0;
      lex_start_pos = 0;
      lex_curr_pos = 0;
      lex_last_pos = 0;
      lex_last_action = 0;
      lex_last_action_simple = dummy_action;
      lex_eof_reached = true }
```

⟨*function* Lexing.lexeme 89d⟩≡ (90d)

```
  let lexeme lexbuf =
    let len = lexbuf.lex_curr_pos - lexbuf.lex_start_pos in
```

```
    let s = Bytes.create len in
    Bytes.unsafe_blit lexbuf.lex_buffer lexbuf.lex_start_pos s 0 len;
    (* pad: at this point why not just use Bytes.sub above? *)
    Bytes.to_string s
```

⟨*function* Lexing.lexeme_char 90a⟩≡ (90d)
```
  let lexeme_char lexbuf i =
    Bytes.get lexbuf.lex_buffer (lexbuf.lex_start_pos + i)
```

⟨*function* Lexing.lexeme_start 90b⟩≡ (90d)
```
  let lexeme_start lexbuf =
    lexbuf.lex_abs_pos + lexbuf.lex_start_pos
```

⟨*function* Lexing.lexeme_end 90c⟩≡ (90d)
```
  let lexeme_end lexbuf =
    lexbuf.lex_abs_pos + lexbuf.lex_curr_pos
```

⟨stdlib/lexing_.ml 90d⟩≡
```
  ⟨copyright ocamllex 8a⟩
  open Stdcompat

  (* coupling: lexbuf and lexing.c lexer_buffer must match! *)
  ⟨type Lexing.lexbuf 12g⟩

  let dummy_action x = failwith "lexing: empty token"


  ⟨type Lexing.lex_tables 71c⟩


  ⟨function Lexing.lex_refill 88c⟩

  ⟨function Lexing.from_function 89a⟩

  ⟨function Lexing.from_channel 89b⟩

  ⟨function Lexing.from_string 89c⟩

  ⟨function Lexing.lexeme 89d⟩

  ⟨function Lexing.lexeme_char 90a⟩

  ⟨function Lexing.lexeme_start 90b⟩

  ⟨function Lexing.lexeme_end 90c⟩

  (******************************************************************************)
  (* Helpers for lexers using the compact code generation method *)
  (******************************************************************************)

  (*less: put lex_tables also here *)

  external engine: lex_tables -> int -> lexbuf -> int = "lex_engine"


  (******************************************************************************)
  (* Helpers for lexers using the simple code generation method *)
  (******************************************************************************)
```

```
let get_next_char lexbuf =
  let p = lexbuf.lex_curr_pos in
  if p < lexbuf.lex_buffer_len then begin
    let c = Bytes.unsafe_get lexbuf.lex_buffer p in
    lexbuf.lex_curr_pos <- p + 1;
    c
  end else begin
    lexbuf.refill_buff lexbuf;
    let p = lexbuf.lex_curr_pos in
    let c = Bytes.unsafe_get lexbuf.lex_buffer p in
    lexbuf.lex_curr_pos <- p + 1;
    c
  end


let start_lexing lexbuf =
  lexbuf.lex_start_pos <- lexbuf.lex_curr_pos;
  lexbuf.lex_last_action_simple <- dummy_action

let backtrack lexbuf =
  lexbuf.lex_curr_pos <- lexbuf.lex_last_pos;
  Obj.magic(lexbuf.lex_last_action_simple lexbuf)
```

D.2 ocamlyacc/

D.2.1 yacc/ast.ml

⟨yacc/ast.ml 91⟩≡

⟨*copyright ocamlyacc* 8b⟩

```
open Stdcompat (* for |> *)

(************************************************************************)
(* Types *)
(************************************************************************)
```

⟨*type* Ast.term*(yacc)* 35c⟩
⟨*type* Ast.nonterm*(yacc)* 35d⟩

⟨*type* Ast.symbol*(yacc)* 35e⟩

⟨*type* Ast.charpos*(yacc)* 35f⟩
⟨*type* Ast.location*(yacc)* 35g⟩
⟨*type* Ast.action*(yacc)* 35h⟩

⟨*type* Ast.grammar*(yacc)* 35a⟩
⟨*type* Ast.rule_*(yacc)* 35b⟩

⟨*type* Ast.directive*(yacc)* 36c⟩

⟨*type* Ast.type_*(yacc)* 36d⟩

⟨*type* Ast.parser_definition*(yacc)* 36b⟩

⟨*constant* Ast.noloc*(yacc)* 35i⟩

```
(* for the augmented grammar *)
```

⟨*constant* `Ast.start_nonterminal` *(yacc)* 37g⟩
⟨*constant* `Ast.dollar_terminal` *(yacc)* 37h⟩

```
(*****************************************************************************)
(* Helpers *)
(*****************************************************************************)
```

⟨*function* `Ast.start_symbol` *(yacc)* 41b⟩

D.2.2 yacc/check.mli

⟨`yacc/check.mli` 92a⟩≡

⟨*type* `Check.error` *(yacc)* 48a⟩

⟨*exception* `Check.Error` *(yacc)* 48b⟩

⟨*signature* `Check.check` *(yacc)* 48c⟩

D.2.3 yacc/check.ml

⟨`yacc/check.ml` 92b⟩≡

```
⟨copyright ocamlyacc 8b⟩
open Ast

(*****************************************************************************)
(* Prelude *)
(*****************************************************************************)

(* TODO:
 * - exist 'start', and 'type' directives
 * - classic use/def: remember set of terms and non terms and look
 *   for use of undefined symbols, or unused symbols.
 * - wrong $ number, too big, $22 not handled for instance
 * - typechecking (but this is done for free by ocaml in the generated code)
 *)

(*****************************************************************************)
(* Types *)
(*****************************************************************************)
```

⟨*type* `Check.error` *(yacc)* 48a⟩

⟨*exception* `Check.Error` *(yacc)* 48b⟩

```
(*****************************************************************************)
(* Helpers *)
(*****************************************************************************)
```

⟨*function* `Check.report_error` *(yacc)* 78b⟩

```
(*****************************************************************************)
(* Main entry point *)
(*****************************************************************************)
```

⟨*function* `Check.check` *(yacc)* 48d⟩

D.2.4 yacc/lr0.mli

⟨yacc/lr0.mli 93a⟩≡

```
⟨type Lr0.ruleidx(yacc) 37e⟩
⟨type Lr0.dotidx(yacc) 49e⟩

⟨type Lr0.stateid(yacc) 36e⟩

⟨type Lr0.item(yacc) 49d⟩

⟨type Lr0.items(yacc) 49c⟩

⟨type Lr0.env(yacc) 37f⟩

⟨type Lr0.automaton(yacc) 49b⟩

⟨signature Lr0.mk_env_augmented_grammar(yacc) 51c⟩

⟨signature Lr0.closure(yacc) 50a⟩

⟨signature Lr0.goto(yacc) 50f⟩

⟨signature Lr0.canonical_lr0_automaton(yacc) 51e⟩

(* helper functions used also by slr.ml *)

⟨signature Lr0.after_dot(yacc) 50c⟩

⟨signature Lr0.all_symbols(yacc) 52a⟩
```

D.2.5 yacc/lr0.ml

⟨yacc/lr0.ml 93b⟩≡

```
⟨copyright ocamlyacc 8b⟩
open Stdcompat (* for |> *)
open Ast

module Set = Set_
module Map = Map_

(**************************************************************************)
(* Prelude *)
(**************************************************************************)
(* Computing the LR(0) automaton for a context free grammar, using
 * the algorithm described in the dragon book in chapter 4.
 *)

(**************************************************************************)
(* Types *)
(**************************************************************************)

⟨type Lr0.ruleidx(yacc) 37e⟩
⟨type Lr0.dotidx(yacc) 49e⟩

⟨type Lr0.stateid(yacc) 36e⟩

⟨type Lr0.item(yacc) 49d⟩
```

⟨*type* `Lr0.items`*(yacc)* 49c⟩

⟨*type* `Lr0.env`*(yacc)* 37f⟩

⟨*type* `Lr0.automaton`*(yacc)* 49b⟩

```
(*****************************************************************************)
(* Helpers *)
(*****************************************************************************)
```

⟨*function* `Lr0.mk_env_augmented_grammar`*(yacc)* 51d⟩

⟨*function* `Lr0.rules_of`*(yacc)* 50e⟩

⟨*function* `Lr0.after_dot`*(yacc)* 50d⟩

⟨*function* `Lr0.move_dot_right`*(yacc)* 51b⟩

⟨*function* `Lr0.all_symbols`*(yacc)* 52b⟩

```
(*****************************************************************************)
(* Algorithms *)
(*****************************************************************************)
```

⟨*function* `Lr0.closure`*(yacc)* 50b⟩

⟨*function* `Lr0.goto`*(yacc)* 51a⟩

```
(*****************************************************************************)
(* Main entry point *)
(*****************************************************************************)
```

⟨*function* `Lr0.canonical_lr0_automaton`*(yacc)* 51f⟩

D.2.6 yacc/first_follow.mli

⟨`yacc/first_follow.mli` 94a⟩≡

⟨*type* `First_follow.first`*(yacc)* 53c⟩

⟨*type* `First_follow.epsilon`*(yacc)* 53d⟩

⟨*signature* `First_follow.compute_first`*(yacc)* 53e⟩

⟨*type* `First_follow.follow`*(yacc)* 55a⟩

⟨*signature* `First_follow.compute_follow`*(yacc)* 55b⟩

D.2.7 yacc/first_follow.ml

⟨`yacc/first_follow.ml` 94b⟩≡

```
⟨copyright ocamlyacc 8b⟩
open Stdcompat (* for |> *)
open Ast
open Lr0 (* for the augmented grammar *)

module Set = Set_
```

```
module Map = Map_

(*****************************************************************************)
(* Prelude *)
(*****************************************************************************)
(* Computing the first and follow set for a context free grammar, using
 * the algorithm described in the dragon book in chapter 4.
 *
 * The only difference with the dragon book is that I've split their
 * FIRST into a first map and an epsilon set, so epsilon is never
 * in first.
 *)

(*****************************************************************************)
(* Types *)
(*****************************************************************************)
```

⟨*type* `First_follow.first`*(yacc)* 53c⟩

⟨*type* `First_follow.epsilon`*(yacc)* 53d⟩

⟨*type* `First_follow.follow`*(yacc)* 55a⟩

```
(*****************************************************************************)
(* Helpers *)
(*****************************************************************************)
```

⟨*function* `First_follow.first_of_sequence`*(yacc)* 56a⟩

⟨*function* `First_follow.epsilon_of_sequence`*(yacc)* 56b⟩

```
(*****************************************************************************)
(* Algorithms *)
(*****************************************************************************)
```

⟨*function* `First_follow.compute_first`*(yacc)* 54⟩

⟨*function* `First_follow.compute_follow`*(yacc)* 55c⟩

D.2.8 yacc/lrtables.ml

⟨`yacc/lrtables.ml` 95a⟩≡

⟨*type* `Lrtables.action`*(yacc)* 37c⟩

⟨*type* `Lrtables.action_table`*(yacc)* 37b⟩

⟨*type* `Lrtables.goto_table`*(yacc)* 37d⟩

⟨*type* `Lrtables.lr_tables`*(yacc)* 37a⟩

D.2.9 yacc/slr.mli

⟨`yacc/slr.mli` 95b⟩≡

⟨*signature* `Slr.lr_tables`*(yacc)* 52c⟩

D.2.10 yacc/slr.ml

⟨yacc/slr.ml 96a⟩≡
```
⟨copyright ocamlyacc 8b⟩
open Stdcompat (* for |> *)
open Ast
open Lr0
open Lrtables

module Set = Set_
module Map = Map_

(******************************************************************************)
(* Prelude *)
(******************************************************************************)
(* Computing the SLR(1) tables for a context free grammar using
 * the algorithm described in the dragon book in chapter 4.
 *)

(******************************************************************************)
(* Helpers *)
(******************************************************************************)

⟨function Slr.filter_some(yacc) 79a⟩

⟨function Slr.map_filter(yacc) 79b⟩


(******************************************************************************)
(* Main entry point *)
(******************************************************************************)

⟨function Slr.lr_tables(yacc) 56c⟩
```

D.2.11 yacc/lalr.mli

⟨yacc/lalr.mli 96b⟩≡

D.2.12 yacc/lalr.ml

⟨yacc/lalr.ml 96c⟩≡

D.2.13 yacc/output.mli

⟨yacc/output.mli 96d⟩≡

```
⟨signature Output.output_parser(yacc) 58a⟩
```

D.2.14 yacc/output.ml

⟨yacc/output.ml 96e⟩≡
```
⟨copyright ocamlyacc 8b⟩
open Stdcompat (* for |> *)
open Ast
open Lr0
open Lrtables
```

```
module Set = Set_
module Map = Map_

(****************************************************************************)
(* Prelude *)
(****************************************************************************)

(****************************************************************************)
(* Helpers *)
(****************************************************************************)
```

⟨*constant* `Output.copy_buffer` *(yacc)* 61a⟩

⟨*function* `Output.get_chunk` *(yacc)* 61b⟩

⟨*function* `Output.copy_chunk` *(yacc)* 61c⟩

⟨*function* `Output.replace_dollar_underscore` *(yacc)* 61d⟩

⟨*constant* `Output.spf` *(yacc)* 79f⟩

⟨*function* `Output.int_of_char` *(yacc)* 62a⟩

⟨*function* `Output.extract_dollars_set` *(yacc)* 62b⟩

```
(****************************************************************************)
(* Main entry point *)
(****************************************************************************)
```

⟨*function* `Output.output_parser` *(yacc)* 58b⟩

D.2.15 yacc/dump.mli

⟨`yacc/dump.mli` 97a⟩≡

⟨*signature* `Dump.dump_item` *(yacc)* 74b⟩

⟨*signature* `Dump.dump_items` *(yacc)* 74c⟩

⟨*signature* `Dump.dump_lr0_automaton` *(yacc)* 74a⟩

⟨*signature* `Dump.dump_lrtables` *(yacc)* 74d⟩

D.2.16 yacc/dump.ml

⟨`yacc/dump.ml` 97b⟩≡

```
⟨copyright ocamlyacc 8b⟩
open Stdcompat (* for |> *)
open Format

open Ast
open Lr0

module Set = Set_
module Map = Map_
```

```
(*****************************************************************************)
(* Prelude *)
(*****************************************************************************)

(* TODO: does not indent things correctly, even after an
 * open_box 2; I don't understand
 *)

(*****************************************************************************)
(* Helpers *)
(*****************************************************************************)

(* common.ml *)
```
⟨*type* `Dump.either`*(yacc)* 79c⟩

⟨*function* `Dump.partition_either`*(yacc)* 79d⟩

⟨*function* `Dump.hash_of_list`*(yacc)* 79e⟩

⟨*function* `Dump.string_of_symbol`*(yacc)* 74g⟩

⟨*constant* `Dump.pf`*(yacc)* 79i⟩
⟨*constant* `Dump.spf`*(yacc)* 79g⟩

⟨*function* `Dump.string_of_action`*(yacc)* 74e⟩

```
(*****************************************************************************)
(* Dumpers *)
(*****************************************************************************)
```

⟨*function* `Dump.dump_symbol`*(yacc)* 74f⟩

⟨*function* `Dump.dump_item`*(yacc)* 75a⟩

⟨*function* `Dump.dump_items`*(yacc)* 75b⟩

⟨*function* `Dump.dump_lr0_automaton`*(yacc)* 75c⟩

⟨*function* `Dump.dump_lrtables`*(yacc)* 76⟩

D.2.17 yacc/tests.ml

⟨yacc/tests.ml 98⟩≡
```
open Stdcompat (* for |> *)
open Ast
open Lr0

module Set = Set_
module Map = Map_
```

⟨*constant* `Tests.arith`*(yacc)* 36a⟩
⟨*function* `Tests.test_lr0`*(yacc)* 49a⟩

⟨*function* `Tests.test_slr`*(yacc)* 52d⟩

⟨*constant* `Tests.arith_ll`*(yacc)* 53a⟩

⟨*function* `Tests.test_first_follow`*(yacc)* 53b⟩

```
open Parsing_
```

⟨*type* `Tests.token`*(yacc)* 38g⟩

⟨*function* `Tests.test_lr_engine`*(yacc)* 38h⟩

D.2.18 yacc/lexer.mll

⟨*backward compatible lexing rules(yacc)* 99a⟩≡ (42a)

```
  (* to be backward compatible with ocamlyacc *)
  and action2 = parse
  | '{'
      { incr brace_depth;
        action2 lexbuf }
  | "%}"
      { decr brace_depth;
        if !brace_depth == 0
        then Lexing.lexeme_start lexbuf
        else action2 lexbuf }
  | "}"
      { decr brace_depth;
        if !brace_depth == 0
        then Lexing.lexeme_start lexbuf
        else action2 lexbuf }
  | "(*"
      { comment_depth := 1;
        comment lexbuf;
        action2 lexbuf }

  | eof { raise (Lexical_error "unterminated action") }
  | _   { action2 lexbuf }

  (* to be backward compatible with ocamlyacc *)
  and comment2 = parse
  | "*/" { () }
  | ['*''/']+ { comment2 lexbuf }
  | eof { raise (Lexical_error "unterminated C comment") }
  | _   { comment2 lexbuf }
```

⟨`Lexer.main()` *backward compatible cases (yacc)* 99b⟩≡ (42c)

```
  (* to be backward compatible with ocamlyacc *)
  | "%%" { main lexbuf }
  | "%{"
      { let n1 = Lexing.lexeme_end lexbuf in
        brace_depth := 1;
        let n2 = action2 lexbuf in
        TAction(Location(n1, n2)) }
  | "/*" { comment2 lexbuf; main lexbuf }
```

D.2.19 yacc/parser.mly

D.2.20 yacc/main.ml

⟨yacc/main.ml 100a⟩≡
```
⟨copyright ocamlyacc 8b⟩
open Ast

(***********************************************************************)
(* Prelude *)
(***********************************************************************)
(* An OCaml port of yacc.
 *
 * The original yacc is written in old C. ocamlyacc in the OCaml
 * distribution is actually also written in C.
 *
 * todo:
 *  - handle priorities, precedences
 *  - EBNF support!
 *)

(***********************************************************************)
(* Main entry point *)
(***********************************************************************)

⟨function Main.main(yacc) 40⟩

⟨toplevel Main._1(yacc) 41a⟩
```

D.2.21 stdlib/parsing_.mli

⟨signature Parsing.symbol_start(yacc) 100b⟩≡ (101h)
```
(* Module [Parsing]: the run-time library for parsers generated by [ocamlyacc]*)

val symbol_start : unit -> int
```

⟨signature Parsing.symbol_end(yacc) 100c⟩≡ (101h)
```
val symbol_end : unit -> int
        (* [symbol_start] and [symbol_end] are to be called in the action part
           of a grammar rule only. They return the position of the string that
           matches the left-hand side of the rule: [symbol_start()] returns
           the position of the first character; [symbol_end()] returns the
           position of the last character, plus one. The first character
           in a file is at position 0. *)
```

⟨signature Parsing.rhs_start(yacc) 100d⟩≡ (101h)
```
val rhs_start: int -> int
```

⟨signature Parsing.rhs_end(yacc) 100e⟩≡ (101h)
```
val rhs_end: int -> int
        (* Same as [symbol_start] and [symbol_end], but return the
           position of the string matching the [n]th item on the
           right-hand side of the rule, where [n] is the integer parameter
           to [lhs_start] and [lhs_end]. [n] is 1 for the leftmost item. *)
```

⟨*signature* Parsing.clear_parser*(yacc)* 101a⟩≡ (101h)
```
val clear_parser : unit -> unit
        (* Empty the parser stack. Call it just after a parsing function
           has returned, to remove all pointers from the parser stack
           to structures that were built by semantic actions during parsing.
           This is optional, but lowers the memory requirements of the
           programs. *)
```

⟨*type* Parsing.parse_tables*(yacc)* 101b⟩≡ (101h)
```
type parse_tables =
  { actions : (parser_env -> Obj.t) array;
    transl_const : int array;
    transl_block : int array;
    lhs : string;
    len : string;
    defred : string;
    dgoto : string;
    sindex : string;
    rindex : string;
    gindex : string;
    tablesize : int;
    table : string;
    check : string;
    error_function : string -> unit }
```

⟨*exception* Parsing.YYexit*(yacc)* 101c⟩≡ (101h)
```
exception YYexit of Obj.t
```

⟨*signature* Parsing.yyparse*(yacc)* 101d⟩≡ (101h)
```
val yyparse :
      parse_tables -> int -> (Lexing.lexbuf -> 'a) -> Lexing.lexbuf -> 'b
```

⟨*signature* Parsing.peek_val*(yacc)* 101e⟩≡ (101h)
```
val peek_val : parser_env -> int -> 'a
```

⟨*signature* Parsing.is_current_lookahead*(yacc)* 101f⟩≡ (101h)
```
val is_current_lookahead : 'a -> bool
```

⟨*signature* Parsing.parse_error*(yacc)* 101g⟩≡ (101h)
```
val parse_error : string -> unit
```

⟨stdlib/parsing_.mli 101h⟩≡
```
(***********************************************************************)
(*                                                                     *)
(*                          Objective Caml                             *)
(*                                                                     *)
(*           Xavier Leroy, projet Cristal, INRIA Rocquencourt          *)
(*                                                                     *)
(*  Copyright 1996 Institut National de Recherche en Informatique et   *)
(*  Automatique.  Distributed only by permission.                      *)
(*                                                                     *)
(***********************************************************************)

  ⟨signature Parsing.symbol_start(yacc) 100b⟩
  ⟨signature Parsing.symbol_end(yacc) 100c⟩
  ⟨signature Parsing.rhs_start(yacc) 100d⟩
  ⟨signature Parsing.rhs_end(yacc) 100e⟩
  ⟨signature Parsing.clear_parser(yacc) 101a⟩

  ⟨exception Parsing.Parse_error(yacc) 38b⟩
```

```
(*--*)

(* The following definitions are used by the generated parsers only.
   They are not intended to be used by user programs. *)

type parser_env
```

⟨*type* Parsing.parse_tables*(yacc)* 101b⟩

⟨*exception* Parsing.YYexit*(yacc)* 101c⟩

⟨*signature* Parsing.yyparse*(yacc)* 101d⟩
⟨*signature* Parsing.peek_val*(yacc)* 101e⟩
⟨*signature* Parsing.is_current_lookahead*(yacc)* 101f⟩
⟨*signature* Parsing.parse_error*(yacc)* 101g⟩

```
(* functions and types used by the generated parsers using the simple code
 * generation method *)
```

⟨*type* Parsing.stateid*(yacc)* 37j⟩
⟨*type* Parsing.nonterm*(yacc)* 37k⟩
⟨*type* Parsing.rule_action*(yacc)* 38c⟩
⟨*type* Parsing.action*(yacc)* 38a⟩

⟨*type* Parsing.lr_tables*(yacc)* 37i⟩

```
type parser_env_simple
```
⟨*type* Parsing.rules_actions*(yacc)* 38d⟩

⟨*signature* Parsing.peek_val_simple*(yacc)* 64a⟩

⟨*signature* Parsing.yyparse_simple*(yacc)* 38e⟩

D.2.22 stdlib/parsing_.ml

⟨*type* Parsing.parser_env*(yacc)* 102⟩≡ (106)

```
(* Internal interface to the parsing engine *)

type parser_env =
  { mutable s_stack : int array;        (* States *)
    mutable v_stack : Obj.t array;      (* Semantic attributes *)

    mutable symb_start_stack : int array; (* Start positions *)
    mutable symb_end_stack : int array;   (* End positions *)

    mutable stacksize : int;            (* Size of the stacks *)
    mutable stackbase : int;            (* Base sp for current parse *)

    mutable curr_char : int;            (* Last token read *)
    mutable lval : Obj.t;               (* Its semantic attribute *)

    mutable symb_start : int;           (* Start pos. of the current symbol*)
    mutable symb_end : int;             (* End pos. of the current symbol *)
```

```
    mutable asp : int;                  (* The stack pointer for attributes *)
    mutable rule_len : int;             (* Number of rhs items in the rule *)
    mutable rule_number : int;          (* Rule number to reduce by *)

    mutable sp : int;                   (* Saved sp for parse_engine *)
    mutable state : int;                (* Saved state for parse_engine *)
    mutable errflag : int }             (* Saved error flag for parse_engine *)
```

⟨*type* Parsing.parse_tables*((stdlib/parsing.ml) (yacc))* 103a⟩≡ (106)

```
  (* coupling: parse_tables and parsing.c parse_tables must match! *)

  type parse_tables =
    { actions : (parser_env -> Obj.t) array;
      transl_const : int array;
      transl_block : int array;
      lhs : string;
      len : string;
      defred : string;
      dgoto : string;
      sindex : string;
      rindex : string;
      gindex : string;
      tablesize : int;
      table : string;
      check : string;
      error_function : string -> unit }
```

⟨*exception* Parsing.YYexit*((stdlib/parsing.ml) (yacc))* 103b⟩≡ (106)

```
  exception YYexit of Obj.t
```

⟨*exception* Parsing.Parse_error*((stdlib/parsing.ml) (yacc))* 103c⟩≡ (106)

```
  exception Parse_error
```

⟨*type* Parsing.parser_input*(yacc)* 103d⟩≡ (106)

```
  type parser_input =
      Start
    | Token_read
    | Stacks_grown_1
    | Stacks_grown_2
    | Semantic_action_computed
    | Error_detected
```

⟨*type* Parsing.parser_output*(yacc)* 103e⟩≡ (106)

```
  type parser_output =
      Read_token
    | Raise_parse_error
    | Grow_stacks_1
    | Grow_stacks_2
    | Compute_semantic_action
    | Call_error_function
```

⟨*constant* Parsing.env*(yacc)* 103f⟩≡ (106)

```
  let env =
    { s_stack = Array.make 100 0;
      v_stack = Array.make 100 (Obj.repr ());
      symb_start_stack = Array.make 100 0;
      symb_end_stack = Array.make 100 0;
      stacksize = 100;
      stackbase = 0;
      curr_char = 0;
```

```
    lval = Obj.repr ();
    symb_start = 0;
    symb_end = 0;
    asp = 0;
    rule_len = 0;
    rule_number = 0;
    sp = 0;
    state = 0;
    errflag = 0 }
```

⟨*function* Parsing.grow_stacks*(yacc)* 104a⟩≡ (106)

```
let grow_stacks() =
  let oldsize = env.stacksize in
  let newsize = oldsize * 2 in
  let new_s = Array.make newsize 0
  and new_v = Array.make newsize (Obj.repr ())
  and new_start = Array.make newsize 0
  and new_end = Array.make newsize 0 in
    Array.blit env.s_stack 0 new_s 0 oldsize;
    env.s_stack <- new_s;
    Array.blit env.v_stack 0 new_v 0 oldsize;
    env.v_stack <- new_v;
    Array.blit env.symb_start_stack 0 new_start 0 oldsize;
    env.symb_start_stack <- new_start;
    Array.blit env.symb_end_stack 0 new_end 0 oldsize;
    env.symb_end_stack <- new_end;
    env.stacksize <- newsize
```

⟨*function* Parsing.clear_parser*(yacc)* 104b⟩≡ (106)

```
let clear_parser() =
  Array.fill env.v_stack 0 env.stacksize (Obj.repr ());
  env.lval <- Obj.repr ()
```

⟨*constant* Parsing.current_lookahead_fun*(yacc)* 104c⟩≡ (106)

```
let current_lookahead_fun = ref (fun (x: Obj.t) -> false)
```

⟨*function* Parsing.yyparse*(yacc)* 104d⟩≡ (106)

```
let yyparse tables start lexer lexbuf =
  let rec loop cmd arg =
    match parse_engine tables env cmd arg with
      Read_token ->
        let t = Obj.repr(lexer lexbuf) in
        env.symb_start <- lexbuf.lex_abs_pos + lexbuf.lex_start_pos;
        env.symb_end   <- lexbuf.lex_abs_pos + lexbuf.lex_curr_pos;
        loop Token_read t
    | Raise_parse_error ->
        raise Parse_error
    | Compute_semantic_action ->
        let (action, value) =
          try
            (Semantic_action_computed, tables.actions.(env.rule_number) env)
          with Parse_error ->
            (Error_detected, Obj.repr ()) in
        loop action value
    | Grow_stacks_1 ->
        grow_stacks(); loop Stacks_grown_1 (Obj.repr ())
    | Grow_stacks_2 ->
        grow_stacks(); loop Stacks_grown_2 (Obj.repr ())
    | Call_error_function ->
        tables.error_function "syntax error";
```

```
        loop Error_detected (Obj.repr ()) in
    let init_asp = env.asp
    and init_sp = env.sp
    and init_stackbase = env.stackbase
    and init_state = env.state
    and init_curr_char = env.curr_char
    and init_errflag = env.errflag in
    env.stackbase <- env.sp + 1;
    env.curr_char <- start;
    try
      loop Start (Obj.repr ())
    with exn ->
      let curr_char = env.curr_char in
      env.asp <- init_asp;
      env.sp <- init_sp;
      env.stackbase <- init_stackbase;
      env.state <- init_state;
      env.curr_char <- init_curr_char;
      env.errflag <- init_errflag;
      match exn with
        YYexit v ->
          Obj.magic v
      | _ ->
          current_lookahead_fun :=
            (fun tok ->
              if Obj.is_block tok
              then tables.transl_block.(Obj.tag tok) = curr_char
              else tables.transl_const.(Obj.magic tok) = curr_char);
          raise exn
```

⟨*function* Parsing.peek_val*(yacc)* 105a⟩≡ (106)

```
  let peek_val env n =
    Obj.magic env.v_stack.(env.asp - n)
```

⟨*function* Parsing.symbol_start*(yacc)* 105b⟩≡ (106)

```
  let symbol_start () =
    if env.rule_len > 0
    then env.symb_start_stack.(env.asp - env.rule_len + 1)
    else env.symb_end_stack.(env.asp)
```

⟨*function* Parsing.symbol_end*(yacc)* 105c⟩≡ (106)

```
  let symbol_end () =
    env.symb_end_stack.(env.asp)
```

⟨*function* Parsing.rhs_start*(yacc)* 105d⟩≡ (106)

```
  let rhs_start n =
    env.symb_start_stack.(env.asp - (env.rule_len - n))
```

⟨*function* Parsing.rhs_end*(yacc)* 105e⟩≡ (106)

```
  let rhs_end n =
    env.symb_end_stack.(env.asp - (env.rule_len - n))
```

⟨*function* Parsing.is_current_lookahead*(yacc)* 105f⟩≡ (106)

```
  let is_current_lookahead tok =
    (!current_lookahead_fun)(Obj.repr tok)
```

⟨*function* Parsing.parse_error*(yacc)* 105g⟩≡ (106)

```
  let parse_error (msg: string) = ()
```

⟨stdlib/parsing_.ml 106⟩≡

```
(**************************************************************************)
(*                                                                        *)
(*                          Objective Caml                                *)
(*                                                                        *)
(*            Xavier Leroy, projet Cristal, INRIA Rocquencourt            *)
(*                                                                        *)
(*  Copyright 1996 Institut National de Recherche en Informatique et      *)
(*  Automatique.  Distributed only by permission.                         *)
(*                                                                        *)
(**************************************************************************)

(* The parsing engine *)

open Stdcompat (* for |> *)
open Lexing

⟨type Parsing.parser_env(yacc) 102⟩

⟨type Parsing.parse_tables((stdlib/parsing.ml) (yacc)) 103a⟩

⟨exception Parsing.YYexit((stdlib/parsing.ml) (yacc)) 103b⟩
⟨exception Parsing.Parse_error((stdlib/parsing.ml) (yacc)) 103c⟩

⟨type Parsing.parser_input(yacc) 103d⟩

⟨type Parsing.parser_output(yacc) 103e⟩

(*
external parse_engine :
    parse_tables -> parser_env -> parser_input -> Obj.t -> parser_output
    = "parse_engine"
*)
let parse_engine a b c d =
  failwith "use yyparse_simple or Parsing module"

⟨constant Parsing.env(yacc) 103f⟩

⟨function Parsing.grow_stacks(yacc) 104a⟩

⟨function Parsing.clear_parser(yacc) 104b⟩

⟨constant Parsing.current_lookahead_fun(yacc) 104c⟩

⟨function Parsing.yyparse(yacc) 104d⟩

⟨function Parsing.peek_val(yacc) 105a⟩

⟨function Parsing.symbol_start(yacc) 105b⟩
⟨function Parsing.symbol_end(yacc) 105c⟩

⟨function Parsing.rhs_start(yacc) 105d⟩
⟨function Parsing.rhs_end(yacc) 105e⟩

⟨function Parsing.is_current_lookahead(yacc) 105f⟩

⟨function Parsing.parse_error(yacc) 105g⟩


(**************************************************************************)
```

```
(* Helpers for parsers using the simple code generation method *)
(**************************************************************************)
```

⟨*type* `Parsing.stateid`*(yacc)* 37j⟩
⟨*type* `Parsing.nonterm`*(yacc)* 37k⟩
⟨*type* `Parsing.rule_action`*(yacc)* 38c⟩

⟨*type* `Parsing.action`*(yacc)* 38a⟩

⟨*type* `Parsing.lr_tables`*(yacc)* 37i⟩

⟨*type* `Parsing.parser_env_simple`*(yacc)* 38f⟩

⟨*type* `Parsing.rules_actions`*(yacc)* 38d⟩

⟨*constant* `Parsing.spf`*(yacc)* 79h⟩

⟨*constant* `Parsing.debug`*(yacc)* 68a⟩
⟨*function* `Parsing.log`*(yacc)* 68b⟩

⟨*function* `Parsing.peek_val_simple`*(yacc)* 64b⟩

⟨*function* `Parsing.value_of_tok`*(yacc)* 64c⟩

⟨*function* `Parsing.yyparse_simple`*(yacc)* 63⟩

Glossary

Indexes

Here is a list of the identifiers used, and where they appear. Underlined entries indicate the place of definition. This index is generated automatically.

Bibliography

[1] Donald E. Knuth. *Literate Programming*. Center for the Study of Language and Information, 1992. For an introduction see http://en.wikipedia.org/wiki/Literate_Program. cited page(s) 7

[2] Yoann Padioleau. Syncweb, literate programming meets unison, 2009. https://github.com/aryx/syncweb. cited page(s) 7