

Exercise 1. Context-free languages

Which of the following are CFLs?

- a $\{a^i b^j \mid i \neq j \wedge i \neq 2j\}$
- b $(a + b)^* = \{(a^n b^n)^n \mid n \geq 1\}$
- c $\{w w^R w \mid w \in (a + b)^*\}$
- d $\{b_i \# b_{i+1} \mid b_i \text{ represents } i \text{ in binary}\}$

Exercise 2. Pushdown automata

Construct a PDA equivalent to the following grammar.

$$S ::= aAA \quad A ::= aS \mid bS \mid a$$

Exercise 3. Pushdown automata

The deterministic PDA (DPDA) is not equivalent to the nondeterministic PDA (NPDA). Consider the language

$$L = \{0^n 1^n \mid n \geq 1\} \cup \{0^n 1^{2n} \mid n \geq 1\}$$

- a Show that L is a CFL.
- b Prove that L is not accepted by a DPDA.

Exercise 4. Regular languages

Show that if L is a CFL over a one-symbol alphabet, then L is regular.

Exercise 5. Laboratory

For this lab, we want to minimize the DFAs generated during lab3, using the minimization algorithm defined by the Myhill-Nerode theorem. You may use the solution code for the *REGEX* \rightarrow *NFA* \rightarrow *DFA* reduction if you like, (to be posted on 10/25/02).

There are two parts to this lab. First, the DFA from lab3 uses a `state_set` to represent a state; we want to translate this to an integer. Second, we will use the Myhill-Nerode to minimize the automaton. In this lab, you will be writing imperative code (code that uses assignment). **Don't** use imperative code in future assignments unless we tell you to.

1 Organizing the code

For this lab, we are finally splitting the functionality into separate files. Here are the files:

File	Contents
nfa.ml	The code from lab2
nfa.mli	Type signatures for nfa.ml
minimize.ml	This is the code you will write
minimize.mli	Type signatures for minimize.ml (provided)
lex.ml	This is the main program
Makefile	This is the configuration file used by the "make" program

The Makefile is used by make to build the program. To build the program, type “make” at a command prompt. For example, here is what appears on my machine kenai.

```
<jyh:kenai 107>make
ocamlc -g -warn-error A -c nfa.mli
ocamlc -g -warn-error A -c nfa.ml
ocamlc -g -warn-error A -c minimize.mli
ocamlc -g -warn-error A -c minimize.ml
ocamlc -g -warn-error A -c lex.ml
ocamlc -g -warn-error A -o lex nfa.cmo minimize.cmo lex.cmo
<jyh:kenai 108>./lex test.txt
Syntax error: char 0 = 'H'
Exit 1
```

The make program builds the program for this lab, called lex. It first compiles the files separately (using the -c option), and then links them (using the -o option). I have a bug in my program, so my test run currently gets a syntax error. A correct program would produce the same output as lab2.

2 Imperative code in OCaml

Imperative code means code that contains assignment functions. You don't have to do it this way, but it will be easiest to use imperative code for this lab. There are three main things you need to know. Read the chapter from the OCaml book (Hickey) on imperative code and assignments.

2.1 Mutable variables

Mutable variables are defined with the **ref** function. You can get the value with the **!** operation, and they are modified with **:=**.

```
# let x = ref 1;;
val x : int ref = {contents = 1}
# !x;;
- : int = 1
# x := 2;;
- : unit = ()
# !x;;
- : int = 2
```

2.2 For loops

For loops are declared with the form:

```
for i = start to stop do
  body
done
```

The code in the body is executed for values of *i* from *start* to *stop*, inclusive.

```

# for i = 0 to 10 do
  Printf.printf "i = %d\n" i
done;;
i = 0
i = 1
i = 2
i = 3
i = 4
i = 5
i = 6
i = 7
i = 8
i = 9
i = 10
- : unit = ()

```

2.3 Arrays

See the Array module for details. You can create an array that contains elements of any type, but the type of all elements in the array must be the same.

- Arrays are created with `Array.create`

```

# let a = Array.create 5 2.2;;
val a : float array = [|2.2; 2.2; 2.2; 2.2; 2.2|]

```

- Use the `a.(i)` operation to get a value.

```

# a.(4);;
- : float = 2.2

```

- Use the `a.(i) <- e` operation to store a value.

```

# a.(4) <- 3.7;;
- : unit = ()
# a.(4);;
- : float = 3.7
# a.(3);;
- : float = 2.2

```

- Matrices are just like arrays of arrays.

```

# Array.make_matrix 2 3 7.1;;
val b : float array array = [| [|7.1; 7.1; 7.1|]; [|7.1; 7.1; 7.1|] |]
# b.(1).(2);;
- : float = 7.1
# b.(1).(2) <- 3.8;;
- : unit = ()
# b;;
- : float array array = [| [|7.1; 7.1; 7.1|]; [|7.1; 7.1; 3.8|] |]

```

3 State renumbering

It will be *much* more convenient to use DFAs that use integer states. Here is the type definition for a DFA from lab2:

```
(* state_set definition *)
type dfa =
  { dfa_delta : state_set -> symbol -> state_set;
    dfa_start : state_set;
    dfa_is_final : state_set -> bool
  }
```

In this definition `state_set` represents a set of integer states. The $\delta(q, c)$ function is the transition function. It takes a `state_set` q , a character c , and produces a new `state_set`. The start state is also a `state_set`.

Again, it would be convenient to use an integer instead of a `state_set`. We would really like to use the following type definition.

```
(* int definition *)
type dfa =
  { dfa_delta : int -> symbol -> int;
    dfa_start : int;
    dfa_is_final : int -> bool
  }
```

Of course, we need the `state_set` representation (for the old code), and the `int` representation (for the new code). To do this, we can use *polymorphism*. Polymorphism lets us define *parameterized* types. Here is the actual type definition for lab4.

```
type 'a dfa =
  { dfa_delta : 'a -> symbol -> 'a;
    dfa_start : 'a;
    dfa_is_final : 'a -> bool
  }
```

The `'a` notation defines a type parameter. To *instantiate* the type, OCaml uses the notation `<type> dfa`. For example, the type `state_set dfa` is equivalent to the “state_set definition,” and `int dfa` is equivalent to the “int definition.”

For this part of the lab, we want to write the following function, defined in `minimize.mli`.

```
(*
 * Convert a DFA using a state_set to a DFA with integer states.
 *)
val int_dfa_of_state_set_dfa : state_set dfa -> int dfa
```

You can implement this function in two ways. One easy way is to define translation functions.

```
int_of_state_set : state_set -> int
state_set_of_int : int -> state_set
```

For example, given a `state_set` with maximum state N , you could define

$$\text{int_of_state_set}(\{q_1, q_2, \dots, q_m\}) = q_1 * N^N + q_2 * N^{N-1} + \dots + q_m * N^{N-m}$$

This is horrible—there are just too many states to represent in a small integer. A better way is to number the *reachable* states. A state q is reachable if there is some string x s.t. $\hat{\delta}(q_0, x) = q$.

Here is a sketch of the code. First, we have the standard header:

```
(*
 * Generate an int dfa from a state_set dfa.
 *)
let int_dfa_of_state_set_dfa (dfa : state_set dfa) : int dfa =
  let { dfa_states = states;
        dfa_delta = delta;
        dfa_start = start;
        dfa_is_final = is_final
      } = dfa
  in
  ...
```

Next, we want to number all the reachable states. The `reachable_states` function has three parameters:

state.table A map `state_set -> int` that represents the current state translation,

state.count The total number of states counted so far

s The current state being examined

The function returns a pair (table, count) where `table` is the translation function, and `count` is the total number of states. Here is the pseudo-code:

```
(* Assign numbers to all the states that are reachable from state s *)
let rec reachable_states state_table state_count s =
  if StateSetTable.mem s state_table then
    (* If we have already assigned a number to this state, don't do anything *)
    state_table, state_count
  else
    (* Otherwise, assign a new number to the state, and add reachable states *)
    let state_table = StateSetTable.add s state_count state_table in
    let state_count = state_count + 1 in

    (* Add all the states reachable in one more step *)
    symbol_fold (fun (state_table, state_count) c ->
      reachable_states state_table state_count (delta s c)) (state_table, state_count)
  in
  ...
```

The `symbol_fold f` function is supposed to call f for each symbol c .

Next, we build a table `int -> state_set` that gives that state for each number.

```
(* Build the inverse map that gives the state for each number *)
let int_table = Array.create state_count StateSet.empty in
let _ =
  StateSetTable.iter (fun s i ->
    int_table.(i) <- s) state_table
in
```

Next, we have to define the δ functions, etc. The new $\delta(q, c)$ function:

- q is an integer; get the `state_set` from the `int_table`
- apply the old δ function, producing a `state_set`,
- return the integer by looking up from the `state_table`.

```
(* New transition relation *)
let delta i c =
  StateSetTable.find (delta int_table.(i) c) state_table
in
```

The final step is to build the new automaton.

```
(* Build the new DFA *)
{ dfa_states = state_count;
  dfa_delta = delta;
  dfa_start = 0;
  dfa_is_final = is_final
}
```

4 The minimization algorithm

For the final part of this lab, we want to write the minimization function. It is most convenient to use the `int dfa` representation to build the matrix of all the states that are not equal.

The pseudo-code looks like this:

```
let minimize_int_dfa (dfa : int dfa) : int dfa =
  let { dfa_states = states;
        dfa_delta = delta;
        dfa_start = start;
        dfa_is_final = is_final
      } = dfa
  in

  (* Allocate the matrix of states that are not equal *)
  let ne_states = Array.matrix states states false in

  (* All final states are not equal to non-final states *)
  for i = 0 to states - 1 do
    for j = 0 to states - 1 do
      ne_states.(i).(j) <- is_final i <> is_final j
    done
  done;

  (* Two states i, j are not equal if  $\delta(i, c) \neq \delta(j, c)$  *)
  let step () =
    for i = 0 to states - 1 do
      for j = 0 to states - 1 do
        if  $\exists c. \text{ne\_states}.\delta(i, c).\delta(j, c)$  then
          ne_states.(i, j) <- true;
      done
    done
  in
```

compute fixpoint of the step function

(Now we have an adjacency matrix of states that are not equal *)
assign a new state number to each equivalence class*

finally, build the new functions, and the new DFA

5 Getting started

Start first on the state-set DFA to int DFA representation. The `lex` function has several options that use can use to print out and examine your results.

```
<jyh:kenai 159>./lex --help
lexical analysis of a file
-v print verbose debugging info
-print_regex print the regular expressions
-print_nfa print the NFA generated from the regex
-print_dfa print the DFA generated from the NFA
-help display this list of options
--help display this list of options
<jyh:kenai 160>./lex -print_dfa test.txt |& head
DFA: DFA {
    states = 4;
    start = 0;
    state[0] {
        ' ' -> 2;
        '0' -> 1;
        'a' -> 1;
        'A' -> 1;
        '/' -> 1;
        '.' -> 1;
    }
    ...
}
```