CMSC 330, summer 2016

Organization of Programming Languages

Project 3 - Regular Expression Interpreter

Due Jun 27, 2016 11:59pm

Introduction

In this project, you will write an OCaml module to implement NFAs and regular expressions.

Getting Started

Downloading

- Download the archive file p3.zip and extract its contents.

Along with files used to make direct submissions to the submit server (submit.jar, .submit , submit.rb), you will find the following project files:

- Your OCaml program
  - nfa.ml
- Public tests
  - public_NFA_closure.ml
  - public_NFA_move.ml
  - public_NFA_accept.ml
  - public_RE_to_str.ml
  - public_RE_to_NFA.ml
  - public_str_to_NFA.ml
  - public_stats.ml
- Expected output for public test
  - public_NFA_closure.out
  - public_NFA_move.out
  - public_NFA_accept.out
  - public_RE_to_str.out
  - public_RE_to_NFA.out
  - public_str_to_NFA.out
  - public_stats.out
- Ruby script to run public tests
  - goTest.rb

To test your implementation, you can execute the public tests from the command line by typing commands like `ocaml public_RE_to_str.ml`, or you can use the `goTest.rb` script. Note that to use goTest.rb you must first edit the file and specify which diff program to use.

For this project you are allowed to use the library functions found in the Pervasives module loaded by default, as well as functions from the List and String modules. As in the previous project, you are not allowed to use imperative OCaml, except for the `next` function to to generate new NFA state numbers. You will receive a 0 for any functions using restricted features - we will be checking your code!

Project Description

Your job is to implement a module `Nfa` that includes an API for implementing both NFAs and regular expressions. The signature and starter implementation for the NFA module is provided. You may *not* change the `NFA` signature in any way, though your implementation may include more types and functions than are listed in the signature. The implementation contains a parser you can use to make it easier to test your implementation. We say more about the parser, below.

Part 1: NFAs

For the first part of this project, you will write a series of functions to implement NFAs using OCaml.

```
module type NFA =
  sig
    type nfa
    type transition = int * char option * int
    val make_nfa : int -> int list -> transition list -> nfa
    val e_closure : nfa -> int list -> int list
    val move : nfa -> int list -> char -> int list
    val accept : nfa -> string -> bool
    val stats : nfa -> int * int * (int * int) list
    ...
end
```

Here are descriptions of the elements of this signature, and what you need to do to implement them:

- `type nfa` - This is an *abstract type* representing NFAs. It is up to you to decide exactly how NFAs are implemented. Since the type is abstract, no client that uses your module will be able to see exactly how they are implemented.

- `type transition = int * char option * int` - This is a (non-abstract) type we've made up for convenience to describe an NFA transition. In the NFAs for this project, states will simply be identified by number. Then `(s0, Some c, s1)` represents a transition from the state numbered `s0` to the state numbered `s1`, via an arc labeled with the character `c`. Notice that the character is optional---the transition `(s0, None, s1)` represents an epsilon transition from `s0` to `s1`.

- `make_nfa : int -> int list -> transition list -> nfa`. This function takes as input the starting state, a list of final states, and a list of transitions, and returns an NFA. Again, it is up to you to decide exactly how NFAs should be implemented, but you probably do not need to do much more than track these three components (the starting state, final states, and transition list). As one example,

  ```
  let m = make_nfa 0 [2] [(0, Some 'a', 1); (1, None, 2)]
  ```

  sets `m` to be an NFA with start state 0, final state 2, a transition from 0 to 1 on character `a`, and an epsilon transition from 1 to 2.

- `e_closure: nfa -> int list -> int list`. This function takes as input an nfa and a list of states. The output will be a list of states (in any order, with no duplicates) that the NFA might be in making zero or more epsilon transitions, starting from the list of initial states given as an argument to e_closure. For example, letting `m` be the NFA above, e_closure would return the following:

  ```
  e_closure m [0]     (* returns [0]   *)
  e_closure m [1]     (* returns [1;2] *)
  e_closure m [2]     (* returns [2]   *)
  e_closure m [0;1]   (* returns [0;1;2] *)
  ```

- `move : nfa -> int list -> char -> int list`. This function takes as input an nfa, a list of initial states, and a character. The output will be a list of states (in any order, with no duplicates) that the NFA might be in after making one transition on the character, starting from one of the initial states given as an argument to move. For example, letting `m` be the NFA above, move would return the following:

  ```
  move m [0] 'a'    (* returns [1] *)
  move m [1] 'a'    (* returns [] *)
  move m [2] 'a'    (* returns [] *)
  move m [0;1] 'a'  (* returns [1] *)
  ```

  Notice that the NFA uses an implicit dead state. If `s` is a state in the input list and there are no transitions from `s` on the input character, then all that happens is that no states are added to the output list for `s`.

- `accept : nfa -> string -> bool`. This function takes an NFA and a string, and returns true if the NFA accepts the string, and false otherwise. You will find functions in the String library to be helpful.

- `stats : nfa -> int * int * (int * int) list`. This function takes an NFA and returns a tuple containing information about the NFA. The tuple looks like this: (total number of states, number of final states, outgoing edge count list) The list is like a map of the number of outgoing edges of a state, to the number of states with that many outgoing edges. For example: (0,1) means that there is 1 state with 0 outgoing edges The list should contain values that are greater than 0, if there are no states that have 3 outgoing transitions, do not to put (3,0) in the list, we will assume that if it's not in the list that the count is 0

An example of this would look like:

```
(3, 1, [(0,1);(1,2)])
```

This NFA has 3 total states, 1 final state, 1 state with 0 outgoing transitions and 2 states with 1 outgoing transition

Note: You need to be a bit careful whenever you combine NFA representations to be sure that state names (i.e., integers) don't conflict. You might use the following internal function as an aid in this process:

- `next : unit -> int` - Return a new integer, different from any values previously returned by `next`. (This function is defined on the OCaml slides.)

Part 2: Regular Expressions

The second part of this project is to implement regular expressions. The signature `NFA` contains the following declarations:

```
module type NFA =
  sig
     ...
    type regexp =
        Empty_String
      | Char of char
      | Union of regexp * regexp
      | Concat of regexp * regexp
      | Star of regexp

    val regexp_to_string : regexp -> string
    val regexp_to_nfa : regexp -> nfa
    ...
end
```

Here `regexp` is an user-defined OCaml variant datatype that represents regular expressions:

- `Empty_String` represents the regular expression recognizing the *empty string* (not the empty set!). Written as a formal regular expression, this would be `epsilon`.
- `Char c` represents the regular expression that accepts the single character `c`. Written as a formal regular expression, this would be `c`.
- `Union (r1, r2)` represents the regular expression that is the union of `r1` and `r2`. For example, `Union(Char 'a', Char'b')` is the same as the formal regular expression `a|b`.
- `Concat (r1, r2)` represents the concatenation of `r1` followed by `r2`. For example, `Concat(Char 'a', Char 'b')` is the same as the formal regexp `ab`.
- `Star r` represents the Kleene closure of regular expression `r`. For example, `Star (Union (Char 'a', Char 'b'))` is the same as the formal regexp `(a|b)*`

For this part of the project you need to implement the following:

- `regexp_to_string : regexp -> string` takes a a regular expression and returns a string for the regular expression in the postfix notation. Postfix notation is a mathematical notation in which every operator follows all of its operands. For example, the infix expression "(3 + (4 * 8)) + ((6 + 7)/5)" is expressed as "3 4 8 * + 6 7 + 5 / +". Using the same idea for regular expressions, the infix regular expression "((a|b)*aba*)*(a|b)(a|b)" can be represented as "a b | * a b a * . . . * a b | a b | . .". You can do this as a postorder DFS traversal over the regexp data structure.
- `regexp_to_nfa : regexp -> nfa` takes a `regexp` and returns an NFA that accepts the same language as the regular expression. Unlike project 2, as long as your NFA accepts the correct language, the structure of the NFA does not matter (since the NFA produced by regexp_to_nfa will only be tested by seeing which strings it accepts).

Provided: Regular Expressions Parser

In the starter files we have provided code that parses strings into `regexp` values. In particular, it implements the following elements of the signature:

```
module type NFA =
  sig
    ...
    val string_to_regexp : string -> regexp
    val string_to_nfa : string -> nfa
    exception IllegalExpression of string
end
```

where

- `string_to_regexp : string -> regexp` takes a string for a regular expression, parses the string, and outputs its equivalent `regexp`. If the parser determines that the regular expression has illegal syntax, it will raise an `IllegalExpression` exception.
- `string_to_nfa : string -> nfa` takes a string for a regular expression, parses the string, converts it into a `regexp`, and transforms it to an nfa, using your `regexp_to_nfa` function. As such, for this functiont to work, your `regexp_to_nfa` function must be working.

Notes about the grammar for regular expressions implemented by the provided parser:

- The regular expressions can contain only `(`, `)`, `|`, `*`, a, b, ..., z and E (for epsilon).
- Note that the precedence for regular expressions are as follows, from highest to lowest:

Precedence	Operator	Description
Highest	( )	parentheses
|	*	closure
v	.	concatenation
Lowest	|	union

- Note that all the binary operators are **right associative**.
- Your function should throw an `IllegalExpression` exception for invalid regular expressions.

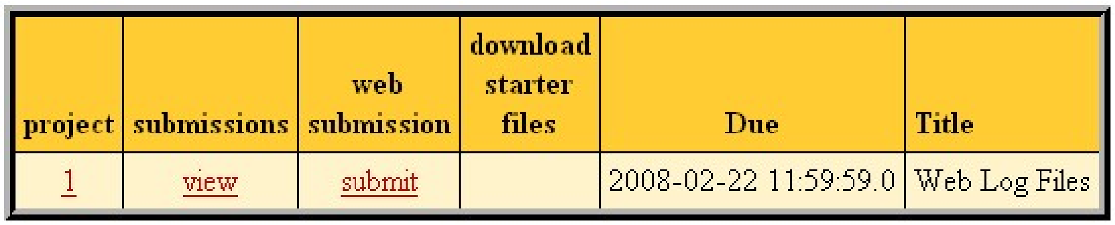
Some examples of regular expressions and their equivalent `regexp` data type are listed in the following table:

String Input	`regexp` Output	String Output
a	`Char 'a'`	a
a|b	`Union(Char 'a',Char 'b')`	a b |
ab	`Concat(Char 'a',Char 'b')`	a b .
aab	`Concat(Char 'a',Concat(Char 'a',Char 'b'))`	a a b . .
(a|E)*	`Star(Union(Char 'a',Empty_String))`	a E | *
(a|E)*(a|b)	`Concat(Star(Union(Char 'a',Empty_String)),Union(Char 'a',Char 'b'))`	a E | * a b | .

Submission

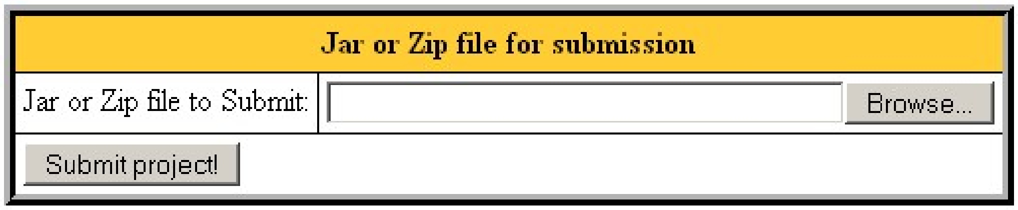
You can submit your project in two ways:

- Submit your file nfa.ml directly to the submit server by clicking on the submit link in the column "web submission".



project	submissions	web submission	download starter files	Due	Title
1	view	submit		2008-02-22 11:59:59.0	Web Log Files

Next, use the submit dialog to submit your nfa.ml file.



Select your file using the "Browse" button, then press the "Submit project!" button.

- Submit directly by executing a Java program on a computer with Java and network access. Use the submit.jar file from the archive p3.zip, To submit, go to the directory containing your project, then either execute submit.rb or type the following command directly:

```
java -jar submit.jar
```

  You will be asked to enter your class account and password, then all files in the directory (and its subdirectories) will be put in a jar file and submitted to the submit server. If your submission is successful you will see the message:

```
Successful submission # received for project 3
```

Academic Integrity

The Campus Senate has adopted a policy asking students to include the following statement on each assignment in every course: "I pledge on my honor that I have not given or received any unauthorized assistance on this assignment." Consequently your program is requested to contain this pledge in a comment near the top.

Please **carefully read** the academic honesty section of the course syllabus. **Any evidence** of impermissible cooperation on projects, use of disallowed materials or resources, or unauthorized use of computer accounts, **will be submitted** to the Student Honor Council, which could result in an XF for the course, or suspension or expulsion from the University. Be sure you understand what you are and what you are not permitted to do in regards to academic integrity when it comes to project assignments. These policies apply to all students, and the Student Honor Council does not consider lack of knowledge of the policies to be a defense for violating them. Full information is found in the course syllabus---please review it at this time.

Web Accessibility