Objectives and Background

In this MP you will learn how to build NDFAs from regular expressions, and use the NDFAs to recognize strings. You will also learn how to share hidden state among a bundle of procedures, which is a precursor of Object-Oriented style of programming. Upon completion of this MP, you should understand:

- Regular expressions
- Finite State Automata
- Sharing hidden state among related procedures

Problem 1

In the next couple of problems you will implement a Regular Expressions engine. The interface to it will consist of two functions.

- (parseRE "regular expression") Will take a regular expression as a string and return a representation of a NDFA.
- (accept ndfa "string") Will recognize the string with the NDFA, and return #t if the string is accepted by the NDFA and #f otherwise.

Regular expressions can consist of:
- strings of characters (concatination)
- union |
- Kleene closure *
- Sub-expressions contained in ()
These problems are a bit involved, so we will provide you with some code to get started, and also a suggested sequence of steps that will help you along. Of course, you do not have to use the code or follow the steps, as long as your program does the right thing. You are not allowed to use existing implementations of regular expressions for scheme within your implementation.

The suggested steps are:

- Make sure you thoroughly understand the kinds of NDFA’s produced by the various RE constructs. So you should be able to take two NDFAs and create a new NDFA that is the union of the two.

- We are providing you with a set of functions that manipulate graphs. Take a look through the starter MP3.scm and the test section of plt/collects/mztake/demos/dijkstra/graph.ss so you know how to work with graphs.

- Create datatype for your NDFA. It will probably consist of a directed graph, along with special markers for the entry node and accept node(s). It would probably be easier if there was only a single accept node in your implementation.

- Write functions for combining NDFAs. They should at least contain the following:
  - (makeNullNdfa) which creates a Ndfa that accepts exactly the empty string.
  - (concatNdfa Ndfa1 Ndfa2 trans) that concatenates 2 Ndfa with the required transition
  - (unionNdfa Ndfa1 Ndfa2)
  - (starNdfa Ndfa)

  Mark your $\epsilon$ edges with a special "epsilon" label.

  We will grade the different functionality of the Ndfa in different problems, so you can start with the most basic (concatination) first, and then add others later.

- Write the (parseRE RE) function. It should convert the string into a list of characters (string-> list) and handle each character individually.

  You should not construct a single NDFA and just modify it as you see each new character, because some characters (*) and |) actually need the preceeding NDFA seperate from all the others. So keep a list of NDFa, and when you reach the end of the RE, concatenate them (in the right order!).

- Write the (accept string) function. Before you do, make sure you understand how your recognition state is represented in the NDFA. You will find (reachable-via-edge) and (successors-via-edge) useful.
For this problem, we will only test concatenation.
Example:

\[
> \text{(define myNdfa1 (parseRE "Hello"))}
> myNdfa1
#<\text{struct:Ndfa}>
> (accept myNdfa1 "Hello")
#t
> (accept myNdfa1 "hello")
#f
> (accept myNdfa1 "Hell")
#f
\]

**Problem 2**

Now add the | Example:

\[
> \text{(define myNdfa2 (parseRE "Wor|uld"))}
myNdfa2
(accept myNdfa2 "World")
(accept myNdfa2 "Would")
(accept myNdfa2 "Woruld")
#<\text{struct:Ndfa}>
#t
#t
#f
\]

\[
> \text{(define myNdfa3 (parseRE "Di|a|ong"))}
myNdfa3
(accept myNdfa3 "Ding")
(accept myNdfa3 "Dang")
(accept myNdfa3 "Dong")
#<\text{struct:Ndfa}>
#t
#t
#t
\]

\[
> \text{(define myNdfa4 (parseRE "Bite|"))}
myNdfa4
(accept myNdfa4 "Bite")
(accept myNdfa4 "Bit")
(accept myNdfa4 "Bitn")
\]
Problem 3

Now add the *:

Example:

```scheme
> (define myNdfa5 (parseRE "He*lp!"))
myNdfa5
(accept myNdfa5 "Hlp!"
(accept myNdfa5 "Heeeeeeelpl!")
#<struct:Ndfa>
#t
#t

> (define myNdfa6 (parseRE "He*|lp!"))
myNdfa6
(accept myNdfa6 "Hlp!"
(accept myNdfa6 "Heeelleeeeelp!"
(accept myNdfa6 "Heeeeeeelpl!")
#<struct:Ndfa>
#t
#f
#t
```

Problem 4

Finally add the sub-expressions (). In this case there is no special function to combine NDFAs into a bigger NDFA. Instead the parser should call itself recursively when it sees a ( and return when it sees a ).

Example:

```scheme
> (define myNdfa7 (parseRE "My name is (Jack)|(Jill)"))
myNdfa7
(accept myNdfa7 "My name is Jack"
(accept myNdfa7 "My name is Jill"
(accept myNdfa7 "My name is JacJill"
#<struct:Ndfa>
#t
```
Problem 5

In this problem you will implement a stack in the style of Section 2.4 in the book.

Implement a procedure create-stack which returns a list of three functions all of which
are bound to some hidden variables. These three functions will be the interface to a stack.

> (define s (create-stack))
> s
 (#:procedure:282:12 #:procedure:283:12 #:procedure:284:12)

The three functions (in order) are:

1. (isEmpty) Should return #t if the stack is empty and #f otherwise.
2. (push x) Should push x onto the stack, return #void.
3. (pop) Should pop the top element off the stack and return it.

The names of the functions are there only as descriptions - you will reference the functions
by their position in the list.

Example:

> ((list-ref s 0))
#t
> ((list-ref s 1) 'hi)
> ((list-ref s 0))
#f
> ((list-ref s 2))
hi
Problem 6

This problem is only required for all the graduate students. Undergraduates may do this problem for extra credit.

Implement the bounded stack variation of the problem above. So you will need a (create-bounded-stack bound) function which again returns a list as in the previous problem. But now (push x) Should push x onto the stack if there only if there is room, and return #t if succeeded and #f if it failed.

Example:

> (define s (create-bounded-stack 2))
> s
> ((list-ref s 0))
#t
> ((list-ref s 1) 'Hi)
#t
> ((list-ref s 1) 'Ahoj)
#t
> ((list-ref s 1) "Guten Tag")
#f
> ((list-ref s 0))
#f
> ((list-ref s 2))
Ahoj
>

Handin

You should hand in a single file named `mp3.scm' with the implementations of the above functions. The names of the functions and number and order of arguments of the functions should be the same as in the problems.

Make sure this assignment runs the way you want on the EWS computers under the MzScheme dialect.

Please see the CS 421 FAQ web page for handin instructions.