Ji Kang

CSS 448

Prof. Jim Hogg

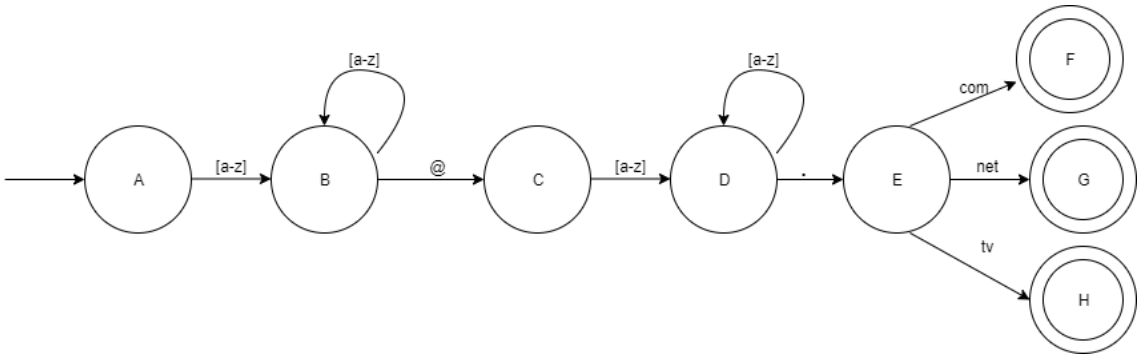
25 January 2019

1. **Draw a DFA that recognizes the simplified email address conforming with the pattern:**

   [a-z]+ @ [a-z]+ (.com | .net | .tv)

Eg: donaldduck@disney.net

Eg: someonefamous@nbc.tv



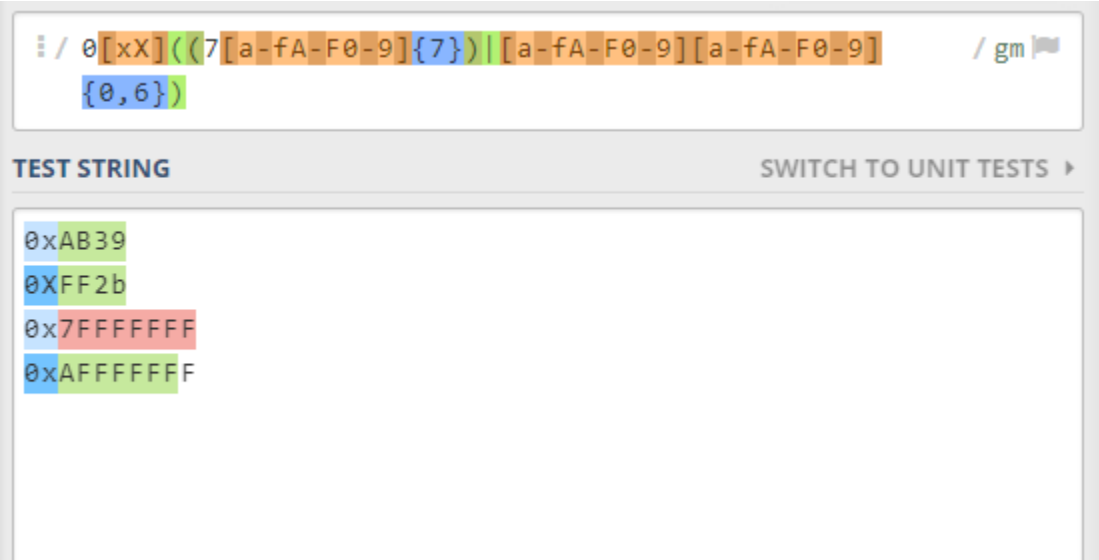
States are labeled A, B, C, D, … and so on.

2. **One of the elements in the definition of a Determinstic Finite Automaton, or DFA, is the transition function:**

δ(s, a) = transition function, where s ∈ S, a ∈ Σ
Explain, in a sentence or two, what the transition function does.

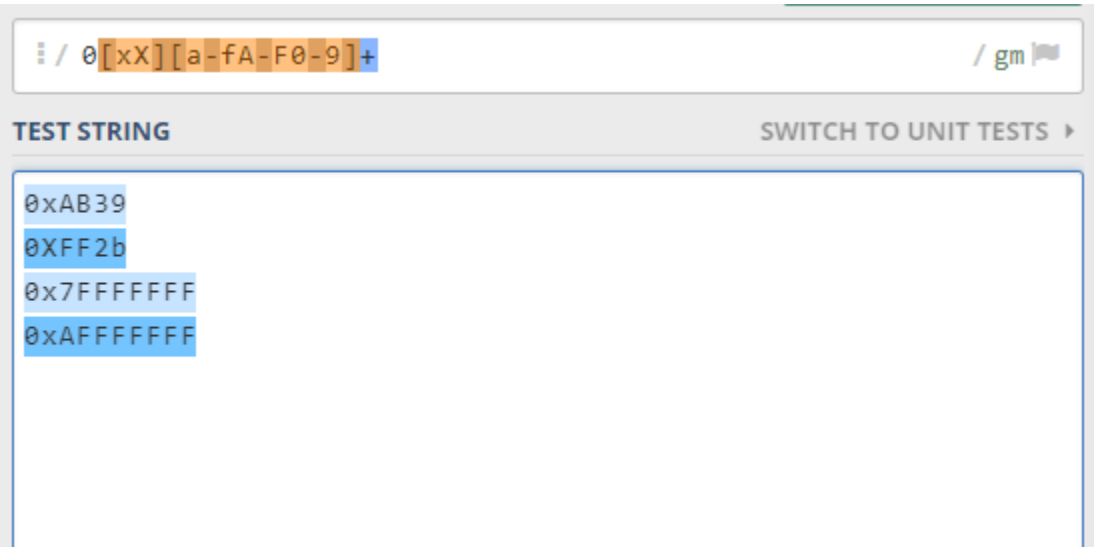
- The transition function takes a state, **s**, which is element in a set of all states **S** and an input in the form of character, **a,** as an input belonging to a set of a finite **alphabet** Σ. Taking these two values, the DFA changes states based on its current state (s) and the input character (a). This state may also be an error state if the character, **a**, is invalid for the given state, **s** (E.g. there are no valid transitions from state, **s**, using input **a**).

3. **Write a regex that describes an integer expressed in hex. Eg: 0xAB39 or 0XFF2b, where lower case or upper case, or a mixture, are allowed**
    1. If we're looking for the regex for a 32-bit signed integer…
        1. Regex = 0[xX]((7[a-fA-F0-9]{7})|[a-fA-F0-9][a-fA-F0-9]{0,6}))



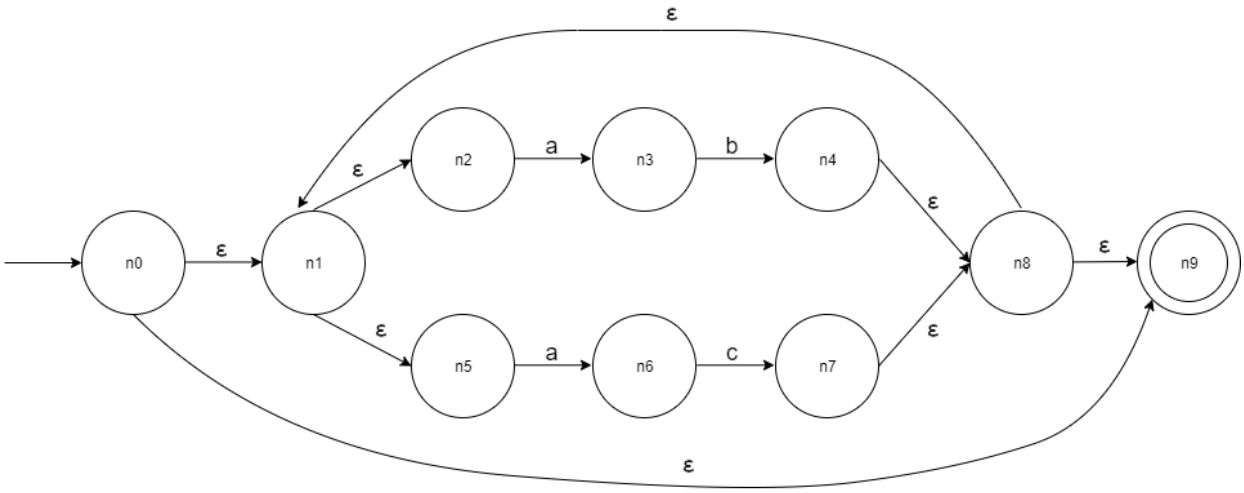
Shows, it'll match all the test examples but won't match a number higher than 7FFFFFFF which is the max for a 32-bit integer (Didn't fully match 0xAFFFFFFF).

2. **If looking for just an arbitrary number in hex**
    1. Regex = 0[xX][a-fA-F0-9]+
        1. Will match any number even it's greater than 2^31 – 1 (0x7FFFFFFF).



Fully matched all hex numbers regardless of the 32-bit limit.

4. Draw the NFA for the regex: (ab | ac)*



5. Convert that NFA to an equivalent DFA. Show both the table used to construct the DFA, as well as a drawing of the DFA

d0 = {n0, n1, n2, n5, n9}

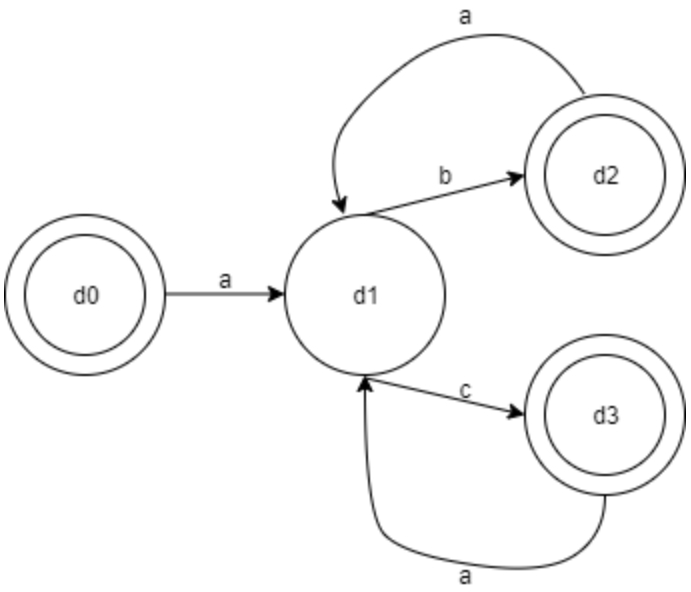
δ(d0, a) = {n3, n6} = d1

δ(d1, b) = {n1, n2, n4, n5, n8, n9} = d2

δ(d1, c) = {n1, n2, n5, n7, n8, n9} = d3

NFA States	**a**	**b**	**c**
d0 = {n0, n1, n2, n5, n9}	d1 = {n3, n6}	None	None
d1 = {n3, n6}	None	d2 = {n1, n2, n4, n5, n8, n9}	d3 = {n1, n2, n5, n7, n8, n9}
d2 = {n1, n2, n4, n5, n8, n9}	d1 = {n3, n6}	None	None
d3 = {n1, n2, n5, n7, n8, n9}	d1 = {n3, n6}	None	None

NFA States	**a**	**b**	**c**
d0	d1	None	None
d1	None	d2	d3
d2	d1	None	None
d3	d1	None	None



6. **Answer the following:**
    a) **"EBNF" : what do the four letters stand for?**
        1. Extended Backus-Naur Form
    b) **What is a Terminal? What is a Non-Terminal? What's another name for a Non-Terminal?**
        1. Terminals are the various inputs we may see belonging to a language. They cannot be broken/derived down into a lesser form. E.g. digits -> [0-9]. Any digit from 0 to 9 is a terminal digit. There exists no rule that would change this character into something else.
        2. Non-terminals are **identifiers** by which the various tokens are defined by the lexical analyzer.
           E.g. in this case, 'digits' is a non-terminal because it has a series of rules that could transform 'digits' into any one of '0' to '9'. They're in a sense "variables". They are restricted to a set of values which may also have another set of values and so on.
    c) **What feature lets us define an infinite language with a finite grammar?**
        1. Recursive definitions of grammars allows us to have an infinite language with a finite set of grammars.
    d) **What is a leftmost derivation?**
        1. Leftmost derivation is deriving the various non-terminal symbols into terminal symbols starting from the leftmost non-terminal.
    e) **What's the difference between a Parse Tree and a Syntax Tree?**
        1. A parse tree maintains all of input given to it as well as the rules/tokens used to match the input text for that given grammar. It keeps track of the input text, the structure of the input, as well as

the non-terminal symbols (identifiers) associated with the tokens belonging to the grammar.

2. A syntax tree (abstract syntax tree) represent the syntactic structure of the input. It doesn't record or use anything from the grammar defined for the input code. So, they're also unaffected by any changes to our input grammar and are much smaller since they do not record the tokens that a parse tree would.

7. **The following Tog program contains at least 5 errors. Refer to the Tog Grammar on the next slide, and write down all the errors you can find in this short program**

```
var a : num
fun main =
  var a:num
  a = 0
  b = 5;
  while a < b
    say a  nl
    a = a + 1
  end
  ret 42
nuf
```

1. Var a : num → No global variable declarations allowed
2. a = 0, b = 5; → To set a variable's value, the "let" keyword needs to be used. Should be "let a = 0" and "let b = 5"
3. b = 5; → No semi-colons in the tog language
4. <in the while-loop>, there is no "do" keyword. The grammar for a while-loop is: "while" Exp "do" Stm+ "elihw". It is missing the terminal "do".
5. a = a + 1 → "let" keyword not used. According to the rule, Let => "let" Nam "=" Exp, there is no "let" before the first 'a'.
6. end → 'end' is not a keyword in tog. To denote the end of a while-loop, we use "elihw".

- Note: "ret 42" is the second to last line. According to the grammar definition for the main function, it cannot have a return type. Rule: "fun" "main" "=" **Body** "nuf".
  - Then in **Body**, the rule is Body => **Var*** **Stm+**
  - The rule for **Stm** is Stm => If | Let | Nl | **Ret** | Say | Stop | While

- Even though the main function can’t return something, the grammar says it’s perfectly legal to have a return statement.
- I wasn’t sure whether this counted as an “error” according to the grammar but as a general rule-of-thumb of programming languages, this wouldn’t be allowed normally.