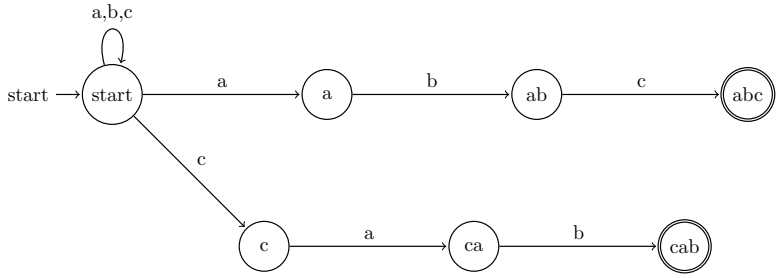
CS 241 – Week 7 Tutorial Solutions

Languages: NFAs and CFGs

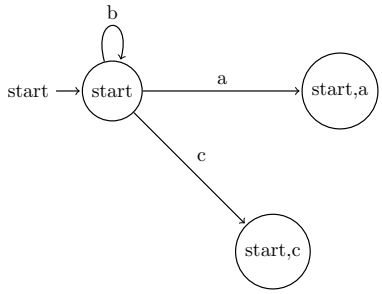
Spring 2015

1 NFA Solutions

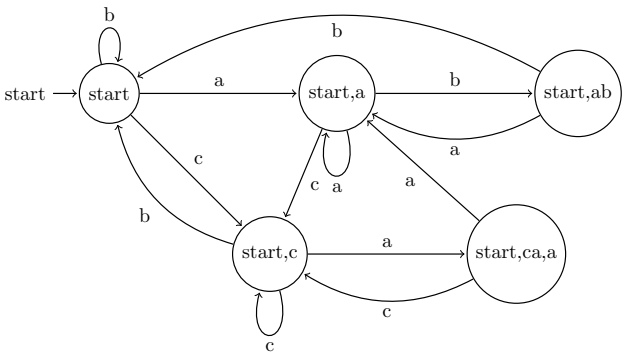
1. We want an NFA for all words that end in abc over the alphabet a,b,c. This is fairly straightforward with an NFA.



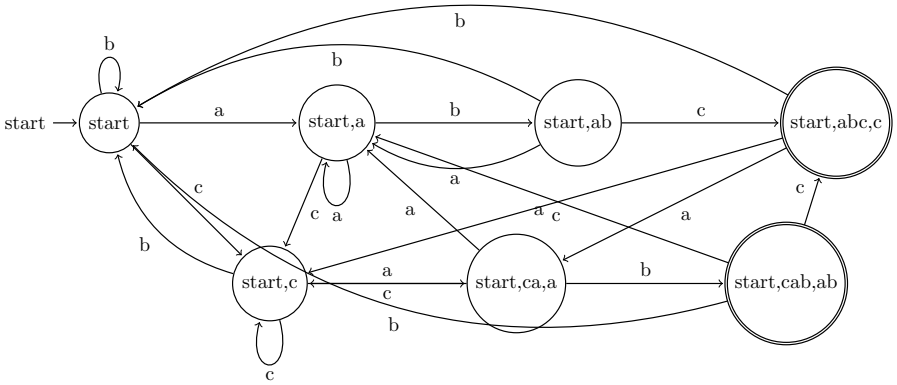
To convert the NFA into a DFA, all we need to do is generate transitions for each letter at each state. Begin with the start state {start}. For each of a, b, c we check the transition function. If the entry corresponds to an existing state, draw an arc for that letter to the existing state. If the state does not exist, create the new state and then draw an arc. After following the three transitions for {start}, the diagram should look like this (note: that we omit the braces in state names to clean things up):



At this point we have two new states that are missing transitions. For both of these states we take the union of the entries shown by the transition function for each of the individual states in the NFA. For example, state {start,a} on letter b would go to {start,ab} because T(start,b) = {start} and T(a, b) = {ab}.After computing all of the transitions for these two states, the diagram should look like this:



This process repeats until no new states are added, and each state in the DFA has a transition for every letter in the alphabet. The last step is to make any state accepting if it contains an accepting state from the original NFA. The final DFA looks like this:



Context-Free Language Solutions

1.

$$S \to AB$$
$$A \to Aa$$
$$A \to Ab$$
$$A \to Ac$$
$$A \to \epsilon$$
$$B \to abc$$
$$B \to cab$$

2.

$$S \to 0S1$$
$$S \to \epsilon$$

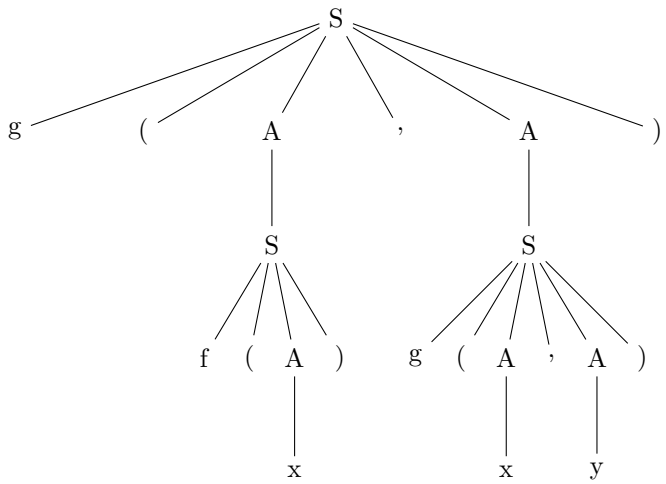
3.

$$S \to aSb$$
$$S \to bSa$$
$$S \to SS$$
$$S \to \epsilon$$

4. A left-canonical derivation for g(f(x),g(x,y)) is:

```
S
⇒ g(A,A)
⇒ g(S,A)
⇒ g(f(A),A)
⇒ g(f(x),A)
⇒ g(f(x),S)
⇒ g(f(x),g(A,A))
⇒ g(f(x),g(x,A))
⇒ g(f(x),g(x,y))
```

The corresponding parse tree is:



5. We can write pseudocode for a tree-based expression evaluation algorithm. The code is straightforward: check what rule was used to expand each node of the tree and return the nesting depth so far, recursing if necessary.

```
eval(tree)
  if (rule is S → S1 S2)
    return max(eval(S1), eval(S2))
  else if (rule is S → ( S ) )
    return 1 + eval(S)
  else if (rule is S → ϵ)
      return 0
```

Now we must figure out how to adapt this to work with the derivation format from .cfg files. Consider this sample input (with the lines numbered):

```
1 S S S
2  S ( S )
3   S ( S )
4    S
5  S ( S )
6   S S S
7    S ( S )
8     S
9    S ( S )
10    S
```

Line 1 corresponds to the root of the parse tree. The child subtree corresponding to the left S corresponds to the next line, line 2. The child subtree corresponding to the right S corresponds to the line after the end of the left S's subtree, which is line 5.

```
eval()
  // consume a line from standard input
  rule ← read_line()
  if (rule is S → S1 S2)
    return max(eval(), eval())
  else if (rule is S → ( S ) )
    return 1 + eval()
  else if (rule is S → ϵ)
      return 0
```

If you manually trace through this function with a sample word, you will see that it produces the correct result.

Note that conditions like "rule is S → (S)" are very general and what we actually want to check for is that rule contains "S (S)", which you will have to figure out how to do in your assignments.