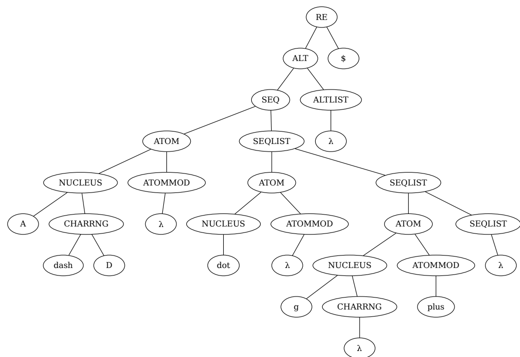
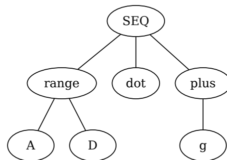
RE Trees to NFAs

A–D.g+



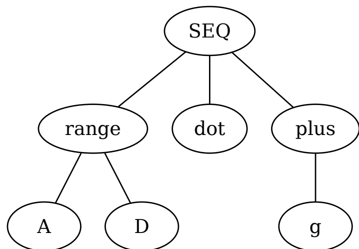
With **syntax directed translation** (*semantic actions*, "production procedures"), we have parsed and verified our RE source (`A–D.g+`) all while coaxing the **concrete syntax tree** ("raw parse tree") into an "RE tree."[1]



The next question is how do we **turn the RE tree into a DFA transition table** for efficient scanning (as in `LUTHOR`).

[1] Because "regular expression expression tree" and "expression tree for the regular expression" seem too, "expressive?"

RE Trees to NFAs

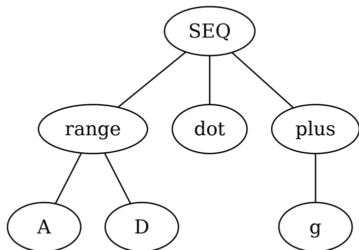


It is tempting to try the conversion from RE tree to optimized DFA (graph or transition table) — but this is a tedious and hard to get right, and by right we mean **verifiably correct**.

(It is reasonable to do this by hand for a small RE, but we need a solution that scales.)

NFAs to the rescue!

RE Trees to NFAs



In general:

- We use a DFS traversal of the RE tree, visiting children in a left-to-right order; a process the book calls a **visitor pattern**.
- Each node of the tree (SEQ, ALT $kleene$, $plus$ and $range$, as well as the leaves dot, λ and $char$) will have a procedure that builds it's part of an NFA.
- After traversal, we can use standard algorithms for converting our NFA to scanner capable DFA transition tables.[1]

[1] Tada! NFAMATCH!

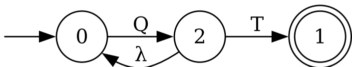
RE Trees to NFAs

Example N and L for RE `Q+T`

T	Q	R	S	T	U
0	2				
+1					
2				1	

L	0	+1	2
0			
+1			
2	•		

Where + means an accepting state, • means $TRUE$.



We will organize the NFA in two parts:

i. A conventional tabular (2d array) form (T). States are rows, characters are columns, and transitions are represented by table cells containing the destination state number.

ii. NFA λ edges will be stored in a square matrix L with rows representing source (originating) state and columns the destination state. There will be no entries along the main diagonal of L.

(How you implement T and L in your project code is up to you).

This is not the T and L that would be generated by `Q+T` when you've completely implemented the ideas in this lecture, it is just an example of the data structures.

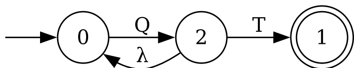
RE Trees to NFAs

Example N and L for RE `Q+T`

T	Q	R	S	T	U
0	2				
+1					
2				1	

L	0	+1	2
0			
+1			
2	•		

Where + means an accepting state, • means $TRUE$.



We also have three functions that manipulate T and L:

a. $addState(T, L)$ adds a new state (a row to T, a row and a column to L) and returns its identifier.

b. $addEdge(T, char, src, dest)$ adds a $char$-edge from state src to state $dest$ in table T.

c. $addLambda(L, src, dest)$ adds a λ-edge from state src to state $dest$ in square matrix L.

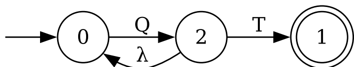
RE Trees to NFAs

Example N and L
for RE `Q+T`

T	Q	R	S	T	U
0	2				
+1					
2				1	

L	0	+1	2
0			
+1			
2	•		

Where + means an accepting state,
• means $TRUE$.



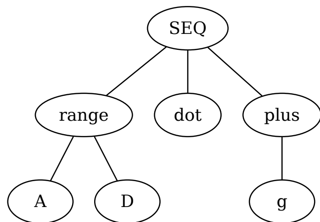
Notice: by the nature of our RE language, we can't represent multiple transitions for one character from a single state **without passing through some sort of language operator, such as parenthetical grouping**, so (if we code our visitor patterns correctly) our T doesn't have to hold this type of information.

We can think of the NFA we are building as being lots of little DFAs connected with λ-edges.

Let's get started!

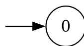
Root of the RE Tree Begins the Traversal

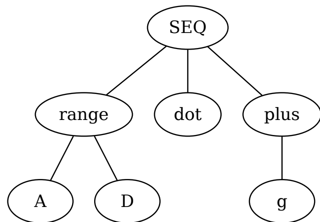
We begin the process initializing T and L which implictly creates the **starting** and **accepting** states 0 and 1.



L	0	+1
0		
+1		

T	A	B	C	D	g
0					
+1					





L	0	+1
0		
+1		

T	A	B	C	D	g
0					
+1					

Root of the RE Tree Begins the Traversal

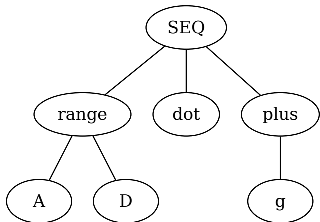
Then we call the visitor pattern function for the root node of our RE Tree with states 0 and 1 as the src and $dest$:

$$nodeSeq(0,1)$$

where $nodeSeq$ is

```
nodeSeq( src, dest ) (
  foreach child from left to right do (
    childdest ← addState(T, L )
    processChild( src, childdest, child )
    src ← childdest
  )
  addLambda( L, childdest, dest )
)
```

$processChild$ calls $child.process$, which is each $child$ node's respective RE Tree to NFA routine, examples of such in these slides are `nodeSeq`, `nodeDot`, `nodeRange` and `nodePlus`.



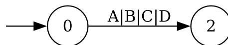
L	0	+1	2
0			
+1			
2			

T	A	B	C	D	g
0	2	2	2	2	
+1					
2					

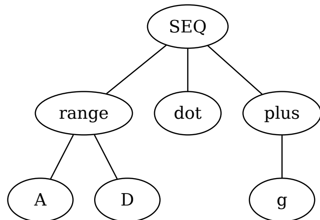
$nodeRange(0,2)$

The first iteration of *nodeSeq* uses *processChild* to call the *range* visitor pattern function with arguments 0 and 2 (2 is the value of *childdest* in *nodeSeq*'s first loop iteration).

```
nodeRange( src, dest ) (
  foreach char ∈ [leftChild,rightChild] ∩ Σ_RE do (
    addEdge(T, char, src, dest )
  )
)
```



1



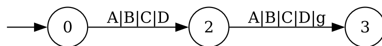
$nodeDot(2,3)$

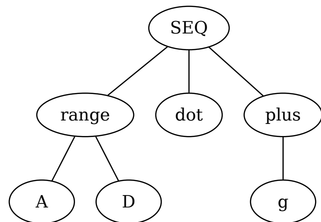
The second iteration of $nodeSeq$'s loop invokes the dot visitor pattern function with arguments 2 and 3 via $processChild$. Notice that the $range$ node's destination state 2 has become the dot node's source state. Not surprising, since we are building a **sequence**.

L	0	+1	2	3
0				
+1				
2				
3				

```
leafDot( src, dest ) (
   foreach char ∈ Σ_RE do (
      addEdge(T, char, src ,dest )
   )
)
```

T	A	B	C	D	g
0	2	2	2	2	
+1					
2	3	3	3	3	3
3					





L	0	+1	2	3	4
0					
+1					
2					
3					
4					

T	A	B	C	D	g
0	2	2	2	2	
+1					
2	3	3	3	3	3
3					
4					

$$nodePlus(3,4)$$

The third (last) iteration of $nodeSeq$'s loop calls the $nodePlus$ visitor pattern function with arguments 3 and 4.

We have several options for implementing $nodePlus$...

L	0	+1	2	3	4	5
0						
+1						
2						
3						
4						
5					•	

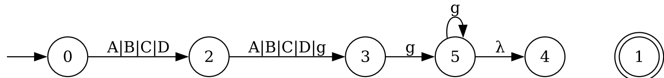
T	A	B	C	D	g
0	2	2	2	2	
+1					
2	3	3	3	3	3
3					5
4					
5					5

$nodePlus(3,4)$

The $nodePlus$ visitor pattern function could be approached in **three** different ways:

First, we could use a brute force solution, (shown in the tables at the left).

Notice how $nodePlus$ added a state (5) in order to manage the Kleene operation. 5 wraps to itself on more gs, and a λ edge provides an "escape hatch" to state 4.



L	0	+1	2	3	4	...
0						
+1						
2						
3						
4						
⋮						

T	A	B	C	D	g
0	2	2	2	2	
+1					
2	3	3	3	3	3
3					?
4					
⋮					

$nodePlus$ via $nodeSeq$ and $nodeKleene$

The second (**better**) way is to develop the interface of your visitor functions so it is easy for them to be used more like building blocks as opposed to single purpose buckets of logic.

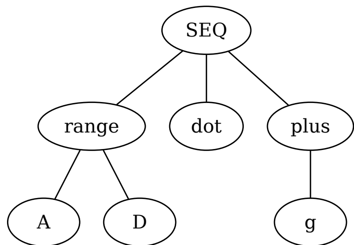
Perhaps your visitor patterns should be implemented in such a way that $nodePlus(src, dest)$ could be written to call

`nodeSeq(` src, x, `List(`$myChildren$`) )`

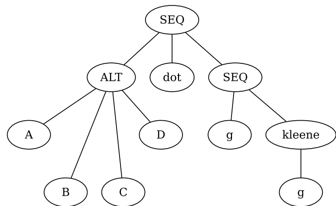
followed by

`nodeKleene(` x, $dest$, `List(`$myChildren$`) )`

for the $nodePlus$ added new state of x (since `A+` $\equiv$ `AA*`).



$\equiv$



nodePlus and (or) *nodeRange* via SDT

The **third way**, also **better than brute force**, probably equal in elegance to a "building blocks" approach to visitor pattern functions, is to construct your SDT routines to change *plus* and *range* trees to their equivilant, more primitive, RE operations:

$$\texttt{A-D} \equiv \texttt{(A|B|C|D)}$$
$$\texttt{a+} \equiv \texttt{aa*}$$

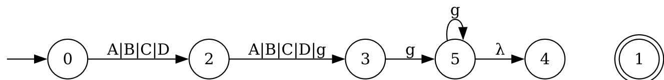
Wrap up the $nodeSeq(0, 1)$ logic after child iterations

L	0	+1	2	3	4	5
0						
+1						
2						
3						
4						
5					●	

T	A	B	C	D	g
0	2	2	2	2	
+1					
2	3	3	3	3	3
3					5
4					
5					5

After we've iterated through all the SEQ node children (the brute force table results of $plus$ are shown), we connect the final sequence's newly allocated $childdest$ to our $dest$ with a λ-edge.

```
nodeSeq( src, dest ) (
  foreach child from left to right do (
    childdest ← addState(T, L )
    processChild( src, childdest, child )
    src ← childdest
  )
  addLambda( L, childdest, dest )
)
```

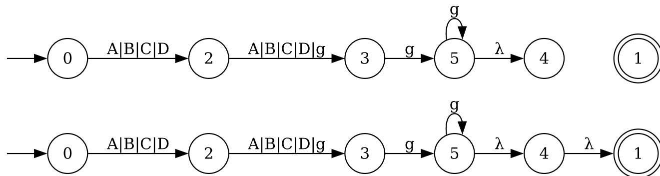


L	0	+1	2	3	4	5
0						
+1						
2						
3						
4		●				
5					●	

T	A	B	C	D	g
0	2	2	2	2	
+1					
2	3	3	3	3	3
3					5
4					
5					5

Wrap up the $nodeSeq(0,1)$ logic after child iterations

After we've iterated through all the SEQ node children (the brute force table results of $plus$ are shown), we connect the final sequence's newly allocated $childdest$ to our $dest$ with a λ-edge.



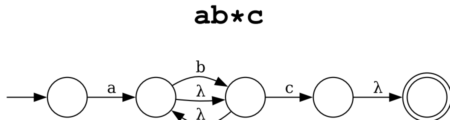
Caveats and Pitfalls

The example logic presented avoided ALT and $Kleene$ representations, these can be a little tricky and require careful thought.

For instance, naive logic for $Kleene$ might look like:

```
nodeKleene( src, dest ) (
   addLambda(L, src , dest )
   addLambda(L, dest , src )
   myChild.process( src, dest )
)
```

This works splendidly for the RE ab*c

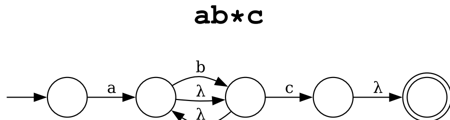


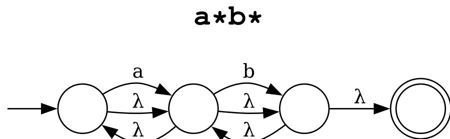
Caveats and Pitfalls

The example logic presented avoided *ALT* and *Kleene* representations, these can be a little tricky and require careful thought.
For instance, naive logic for *Kleene* might look like:

```
nodeKleene( src, dest ) (
   addLambda(L, src , dest )
   addLambda(L, dest , src )
   myChild.process( src, dest )
)
```

This works splendidly for the RE `ab*c`, but fails miserably for `a*b*`.
Why?





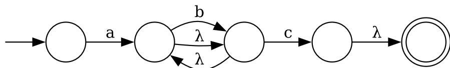
Permits `abab`!

Caveats and Pitfalls

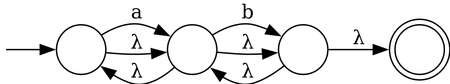
The *src* and *dest* states of `nodeKleene` are not under `nodeKleene`'s "control" — they're provided by the parent RE tree node, and thus the parent can permit other nodes to add incident edges to them.

In this case, the `b*` was permitted to add its "back λ-edge" to `a*`'s *dest*, which created an erroneous path permitting `abab` to be matched by the NFA.

ab*c



a*b*



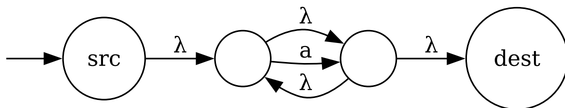
Permits `abab`!

Good Kleene Fun

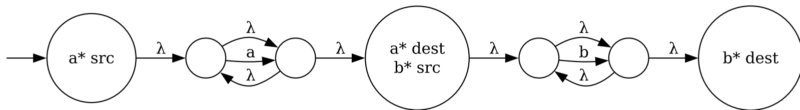
A helpful invariant to keep in mind:

Any node generating "back edges" or "skipping edges" should take care that no other state can connect to the state beginning its DFA logic, or the state terminating its DFA logic.

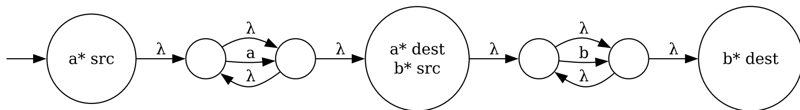
The automata `nodeKleene` **should be building** for `a*` is more like:



Which would yield the correct automata for `a*b*`:



$$processChild \equiv lambdaWrap(src, dest, child)$$



My recommended approach is to use a $lambdaWrap$-ing implementation for $processChild$. $lambdaWrap$ creates "sentinel λ-nodes" around each construction step of the NFA.

I choose to do this for all construction steps — $lambdaWrap$-ing around **any** type of child node could be described as "overly pedantic."

But **overly pedantic** isn't the same as "premature optimization" (IMHO).

One means "more correct than it needs to be", while the other means "more complex than it needs to be".

And I hope you'd agree that the latter is a tried and true design philosophy of engineering (aka Occam's razor), the former is simply non-sensical to a right-minded engineer.

My actual *nodeSeq* implementation

For completeness and clarity, below is the actual implementation I use for *nodeSeq*.

```
nodeSeq( src, dest ) (
  foreach child from left to right do (
    childdest ← addState(T, L )
    lambdaWrap( src, childdest, child )
    src ← childdest
  )
  addLambda( L, childdest, dest )
)
```

I've learned that introducing *lambdaWrap* too early in the explanation of RE→NFA tends to make the whole lecture less than optimal.