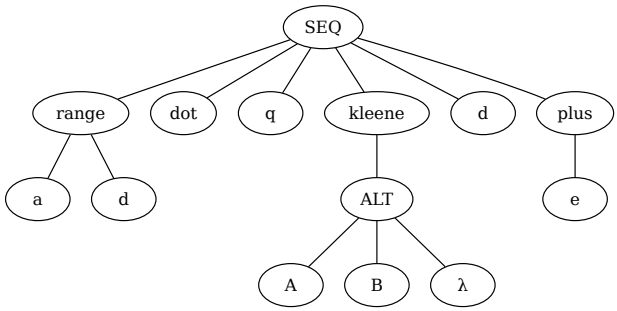
1. Similar to our discussion today in lecture, the regular expression `a-d.q(A|B|)*de+` from the LL(1) grammar below would yield the "raw" or concrete parse tree on page 5.

#	Rules
1	$RE \rightarrow ALT\ \$$
2	$ALT \rightarrow SEQ\ ALTLIST$
3	$ALTLIST \rightarrow pipe\ SEQ\ ALTLIST$
4	$ALTLIST \rightarrow \lambda$
5	$SEQ \rightarrow ATOM\ SEQLIST$
6	$SEQ \rightarrow \lambda$
7	$SEQLIST \rightarrow ATOM\ SEQLIST$
8	$SEQLIST \rightarrow \lambda$
9	$ATOM \rightarrow NUCLEUS\ ATOMMOD$
10	$ATOMMOD \rightarrow kleene$
11	$ATOMMOD \rightarrow plus$
12	$ATOMMOD \rightarrow \lambda$
13	$NUCLEUS \rightarrow open\ ALT\ close$
14	$NUCLEUS \rightarrow char\ CHARRNG$
15	$NUCLEUS \rightarrow dot$
16	$CHARRNG \rightarrow dash\ char$
17	$CHARRNG \rightarrow \lambda$

During the parse several steps can be performed to simplify the structure of the tree, resulting in:



In lecture (`show_re-sdt.pdf`) we discussed the **syntax directed translation** procedures that would be executed on a subset of the grammar rules. This question asks you to compose the pseudo code for the remaining production rules that didn't have their logic presented in lecture.

For guidance, you can use this visualization of `the parsing steps` for this question's example.

If you missed lecture, there is a lengthy explanation of the mechanics of **syntax directed translation** on page 3.

2. We presume your group has by now at least one working LL(1) parser from `ll1-parsing.pdf` . Take any of the many simple grammars we've had in lecture up until now (or one of your own design) and use it to develop a framework in the code base for attaching **SDT procedures** to particular production rules and have them executed (as shown in lecture) when end-of-production markers are encountered in the LL(1) parsing stack.

   Develop for your group a working example that demonstrates at least following:

   a. Flip flops or rotates grammar symbols of a particular production rule's RHS.
   For instance, the grammar rule is $A \rightarrow x\,Y\,z$, but the AST children of A are in the order $Y\,z\,x$ or $z\,x\,Y$ or $z\,Y\,x$.

   b. "Flattens" a recursive rule (such as what happens to *SEQLIST* in the lecture example.
   So grammar rules $B \rightarrow g\ B$ and $B \rightarrow g$ yield an AST node B with many g children, instead of most[1] B nodes having two children g and B.

> Keep in mind, there is no practical use of option (a) above in the parsing of languages, it is simply a useful manipulation that is easy to code and verify in your implementation.

3. Same as question 2 **iff** your group has *two different code implementations*. Otherwise, you can ignore this question.

[1] . . . all but the recursion terminating substitution.

4. Construct a scanning function for our regular expression language on the preceding page and reproduced below for your convenience.[2] It should generate output as sequences of pairwise tuples (not surprisingly very similar to the token output of the `LUTHOR` programming project). From a regular expression such as

```
Ab(cd-e+)*(.|012)3
```

your scanning logic should generate the **token stream** below:

```
char   A
char   b
open   (
char   c
char   d
dash   -
char   e
plus   +
close  )
kleene *
open   (
dot    .
pipe   |
char   0
char   1
char   2
close  )
char   3
```

RE → *ALT* $
ALT → *SEQ ALTLIST*
ALTLIST → *pipe SEQ ALTLIST* | λ
SEQ → *ATOM SEQLIST* | λ
SEQLIST → *ATOM SEQLIST* | λ
ATOM → *NUCLEUS ATOMMOD*
ATOMMOD → *kleene* | *plus* | λ
NUCLEUS → *open ALT close* | *char CHARRNG* | *dot*
CHARRNG → *dash char* | λ

Recall that a scanner generates only the terminals of a language — it will never output a *NUCLEUS*, these are formed by the parser when recognizing patterns in the terminals.

We need to be able to write regular expressions with the *char* data of asterisk (`*`), so we must have a method for **escaping** the special symbols of our RE language. We'll use the ubiquitous backslash (0x5c) for this purpose. It will be helpful to have a few other specially escaped characters as well, here is the full list:

RE sequence	token stream	RE sequence	token stream
\\|	*char* |	*	*char* *
\+	*char* +	\.	*char* .
\(	*char* (	\)	*char*)
\-	*char* -	\s	*char* x20
\n	*char* x0a	\\	*char* \

When turning a regular expression into a token stream, these backslash forms should emit a simple *char* terminal. For example, in the margin at the right is the expected output from a more complex regular expression:

```
\(paren(thetical|)\-ally\)\scorrect!\n
```

```
char   (
char   p
char   a
char   r
char   e
char   n
open   (
char   t
char   h
char   e
char   t
char   i
char   c
char   a
char   l
pipe   |
close  )
char   -
char   a
char   l
char   l
char   y
char   )
char   x20
char   c
char   o
char   r
char   r
char   e
char   c
char   t
char   !
char   x0a
```

Now here's the (small) kicker: Don't implement any of the scanning logic using your language's built-in regex features! We want to finish the course having **bootstrapped** our own REs and Lexer. We can't say that if you use *a regular expression library* from another source. In truth this requirement is not difficult to meet,[3] none of the sequences above require looking ahead more than two characters at a time.

[2] This logic is referred to in `WRECK` as the "silly lexer," because it is quite barely a lexer.
[3] A loop and one state variable?

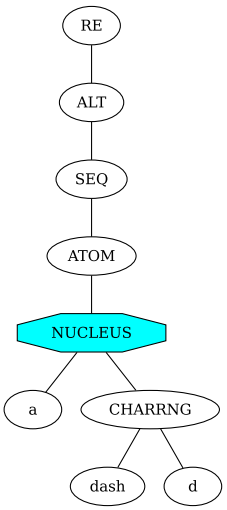
Syntax Directed Translation (SDT) in brief

The thought exercise for question 1 has you determine the algorithms for massaging the raw parse tree (on page 5, also called a ***concrete syntax tree***) into the simplified tree above.

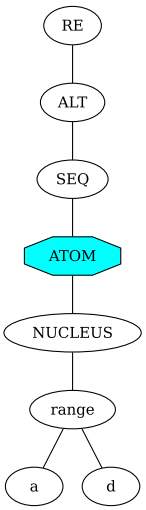
Envision an algorithm where a **rule or non-terminal specific** function is invoked when each internal (non-terminal) node of a parse tree is completed. In this case *completed* means that no more children will be added to the node **and** no more children will be added to any descendants of the node. In the case of the LL parsing algorithm, this is everytime a production marker (`*`) is popped off the stack. See `show_re-sdt.pdf` and `the parsing steps` for a visual of the parsing algorithm.

By way of example, consider how the first *NUCLEUS* node in the left-most derivation of `a-d.q(A|B|)*de+` might be simplified immediately after the first application of rule 14 is completed. The regular expression part is `a-d`, which from the grammar above yields the partial[4] parse tree with the *NUCLEUS* node at the left:

Pre **Post**




```
if ( node.CHARRNG.char exists ) then (
  let rangeNode ← new range node
  rangeNode.addChild(node.char)
  rangeNode.addChild(node.CHARRNG.char)
  replace node.children with rangeNode in parse tree T
  return
)
```



The pseudo code in the middle shows how the parse tree *NUCLEUS* node can be immediately simplified, with the resulting parse tree **Post** to the right. Some points for clarification:

i. **Not all non-terminal nodes require simplification functions** — truly one skill to be developed here is choosing the nodes for simplification in order to *easily* arrive at a simplified parse tree.

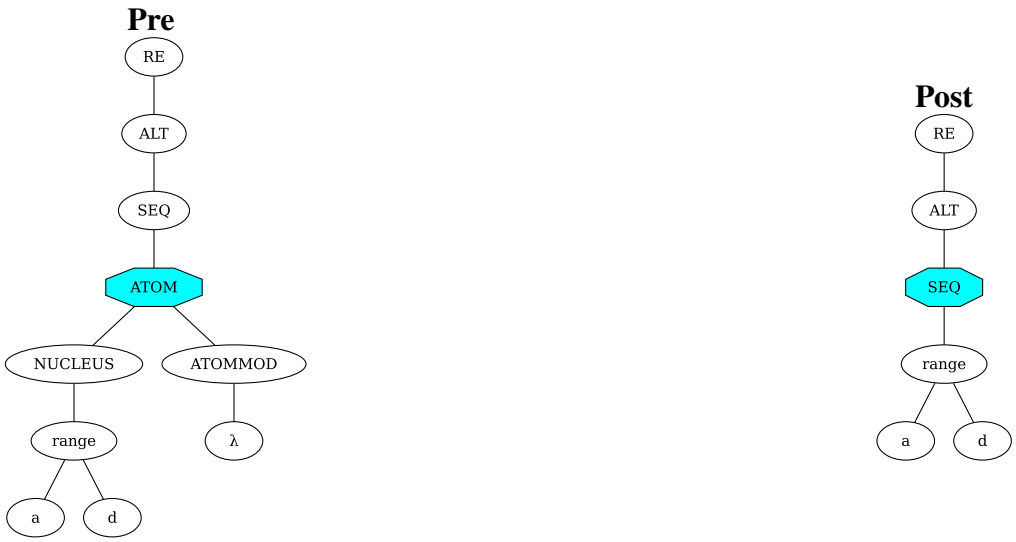
ii. The psuedo code uses terms directly from the grammar and the (partial) raw parse tree. How was the "path" of *node.CHARRNG.char* chosen in the logic? Because *node* ≡ *NUCLEUS* and *node.CHARRNG.char* is the most **convenient and unambiguous** identifier for the second child of *node*'s second child.

iii. The node type of **range** is not a grammar symbol, but that's OK — the input so far has been syntax verified, so we don't need to maintain the originally parsed structure of the input.

iv. Since we consider this simplification occuring as soon as a non-terminal node is *completed*, the tree does not yet show the *ATOMMOD* node deriving to λ and being appended as the right most child of the *ATOM* node now containing **range**.

v. We can envision the **next simplification function** called will be the one for *ATOM* after production rule 9 is completed. The partial parse tree at that time is shown on the following page; since there is no + or * operator applied to **range** the *ATOM* can be "simplified away" leaving **range** as the first child of *SEQ*.

[4] It is a **partial** parse tree because it is still being constructed by an LL (top-down) parse — the only place you see full, raw parse trees are in university compilers courses — in practice simplification of the parse tree occurs hand-in-hand and concurrently with the parse!



Your task for question(s) 1 of this LGA: design the required simplification logic for transforming the (partial) LL(1) generated parse trees created during the recursive descent parse into a simplifed AST (the tree on on page 1).

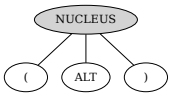
For completeness, we show the complete pseudo code for simplifying *NUCLEUS* nodes (grammar rules 13–15):

```
procedure NUCLEUS( node, parse tree T )

  if ( node.children[0] is open ) then (
    replace node with node.children[1] in parse tree T
    return
  )

  if ( node.CHARRNG.char exists ) then (
    let rangeNode ← new range node
    rangeNode.addChild(node.char)
    rangeNode.addChild(node.CHARRNG.char)
    replace node.children with rangeNode in parse tree T
    return
  )

  if ( node.CHARRNG.child is λ ) then (
    remove node.CHARRNG child
    return
  )

  if ( node.child is dot ) then (
    replace node with node.child in parse tree T
    return
  )
```

Beginning on the following page are the transformations of the parse (sub)trees that occur depending on the production rule being processed. In all PRE trees, the grey node represents *node* of the *NUCLEUS* logic; in the POST **the only colored node** represents the **the same location in the parse tree** as the PRE grey *node*.

Rule 13, *NUCLEUS* → *open ALT close*

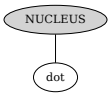
PRE



POST

Rule 15, *NUCLEUS* → *dot*

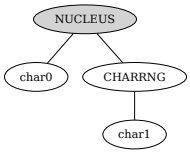
PRE



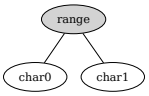
POST

Rule 14 when *CHARRNG.char* exists

PRE

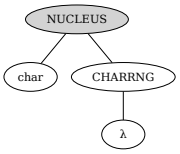


POST



Rule 14 when *CHARRNG.child* is λ

PRE



POST

"Raw" Parse Tree for input `a-d.q(A|B|)*de+`, also called a *concrete parse tree*