Compiling a Regular Expression

$$
\begin{array}{rcl}
RE & \rightarrow & ALT\ \$ \\
ALT & \rightarrow & SEQ\ ALTLIST \\
ALTLIST & \rightarrow & pipe\ SEQ\ ALTLIST \\
 & | & \lambda \\
SEQ & \rightarrow & ATOM\ SEQLIST \\
 & | & \lambda \\
SEQLIST & \rightarrow & ATOM\ SEQLIST \\
 & | & \lambda \\
ATOM & \rightarrow & NUCLEUS\ ATOMMOD \\
ATOMMOD & \rightarrow & kleene \\
 & | & plus \\
 & | & \lambda \\
NUCLEUS & \rightarrow & open\ ALT\ close \\
 & | & char\ CHARRNG \\
 & | & dot \\
CHARRNG & \rightarrow & dash\ char \\
 & | & \lambda
\end{array}
$$

To the left is an LL(1) compatible grammar for simple regular expressions, **sufficient for lexing** (scanning) simple program source. Notice the **order of operations** within the grammar:

- The lowest precedence operator is **ALT**ternation; it is also wrapped by parenthesis

$$NUCLEUS \rightarrow open\,ALT\,close$$

  which makes the form highest precedence.
- ALT s are made up of one or more **SEQ**uences, separated by the $pipe$ symbol,
- and SEQ s are a list of **ATOM**s

Compiling a Regular Expression

$$
\begin{array}{rcl}
RE & \rightarrow & ALT\ \$ \\
ALT & \rightarrow & SEQ\ ALTLIST \\
ALTLIST & \rightarrow & pipe\ SEQ\ ALTLIST \\
 & | & \lambda \\
SEQ & \rightarrow & ATOM\ SEQLIST \\
 & | & \lambda \\
SEQLIST & \rightarrow & ATOM\ SEQLIST \\
 & | & \lambda \\
ATOM & \rightarrow & NUCLEUS\ ATOMMOD \\
ATOMMOD & \rightarrow & kleene \\
 & | & plus \\
 & | & \lambda \\
NUCLEUS & \rightarrow & open\ ALT\ close \\
 & | & char\ CHARRNG \\
 & | & dot \\
CHARRNG & \rightarrow & dash\ char \\
 & | & \lambda
\end{array}
$$

To the left is an LL(1) compatible grammar for simple regular expressions, **sufficient for lexing** (scanning) simple program source. Notice the **order of operations** within the grammar:

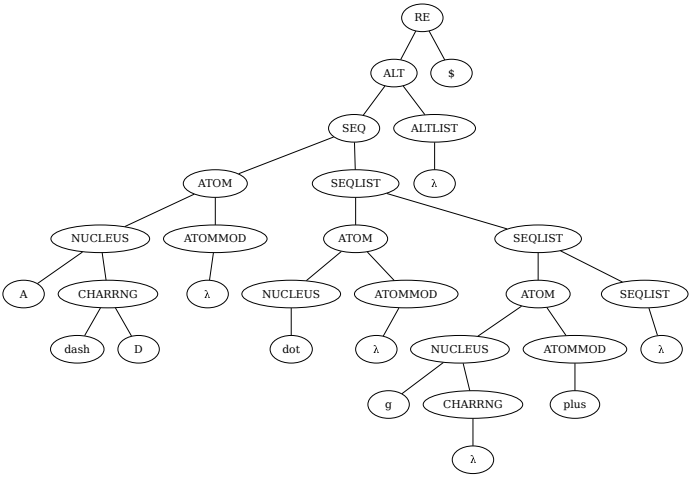
- An $ATOM$ is made up of its **NUCLEUS** and possibly an operator: **kleene** (`*` , any number of) or **plus** (one or more).
- A $NUCLEUS$ may be either a parenthetical group (`(a|bc)*`), a character range (just `t` or `Q-V`), or the any-character dot (`.`)

The Concrete ("Raw") Parse Tree

Input RE: **`A-D.g+`**

The raw (**concrete**) parse tree result shows there are **no syntax errors** in the source regular expression. But compared to its **simplified abstract syntax tree** it has several downsides:

1. Large (as in a much larger data structure),
2. Most of the internal nodes are not needed to represent the **semantic meaning** of the original regex,
3. Did we mention **big**?

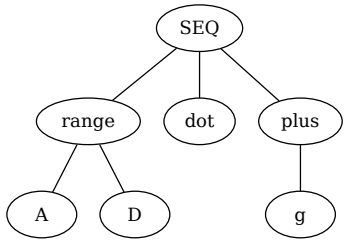


An *Abstract Syntax Tree* (AST)

We would like an efficient process for translating raw parse trees to ASTs, ideally we could do this **during the parse**.

The **simplified AST** is $< 25\%$ the size of the original version, and this is for a very small input.

Input RE: **`A-D.g+`**



Operation: begin

STACK

QUEUE

PARSE TREE

Input RE: **A–D.g+**

Operation: `char` predicts rule 1 $RE \rightarrow ALT\ \$$

STACK

QUEUE

PARSE TREE

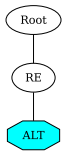
Input RE: **`A-D.g+`**

Operation: `char` predicts rule 2 $ALT \rightarrow SEQ\ ALTLIST$

STACK

QUEUE

PARSE TREE



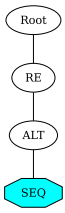
Input RE: **`A-D.g+`**

Operation: `char` predicts rule 5 $SEQ \rightarrow ATOM\ SEQLIST$

STACK

QUEUE

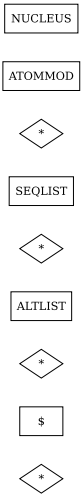
PARSE TREE



Input RE: **`A-D.g+`**

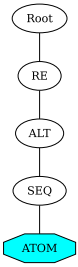
Operation: `char` predicts rule 9 $ATOM \rightarrow NUCLEUS\ ATOMMOD$

STACK



QUEUE

PARSE TREE



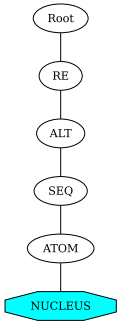
Input RE: **`A-D.g+`**

Operation: `char` predicts rule 14 $NUCLEUS \rightarrow char\ CHARRNG$

STACK

QUEUE

PARSE TREE

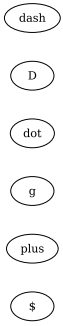


Input RE: **`A-D.g+`**

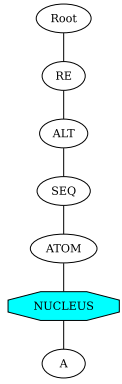
Operation: token `char` match (`char`)

STACK

QUEUE



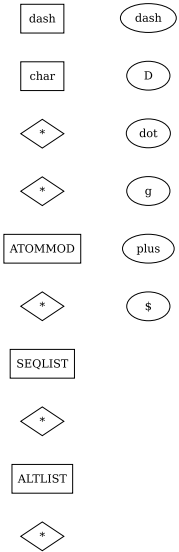
PARSE TREE



Input RE: **`A–D.g+`**

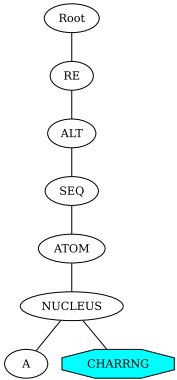
Operation: `dash` predicts rule 16 $CHARRNG \rightarrow dash\ char$

STACK



QUEUE

PARSE TREE



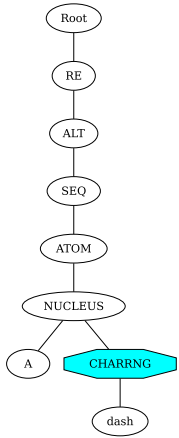
Input RE: **`A-D.g+`**

Operation: token `dash` match (`dash`)

STACK

QUEUE

PARSE TREE

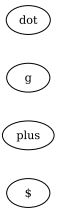


Input RE: **`A-D.g+`**

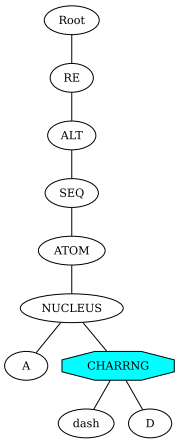
Operation: token `char` match (`char`)

STACK

QUEUE



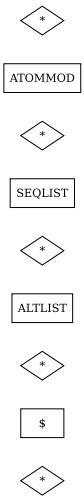
PARSE TREE



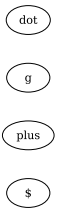
Input RE: **`A-D.g+`**

Operation: end of *CHARRNG* production

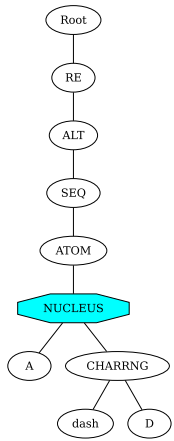
STACK



QUEUE



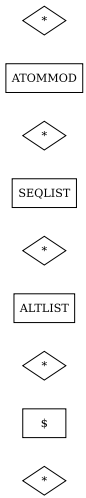
PARSE TREE



Input RE: **`A-D.g+`**

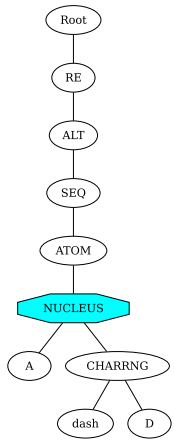
Operation: end of $NUCLEUS$ production — before rule 14 SDT procedure

STACK



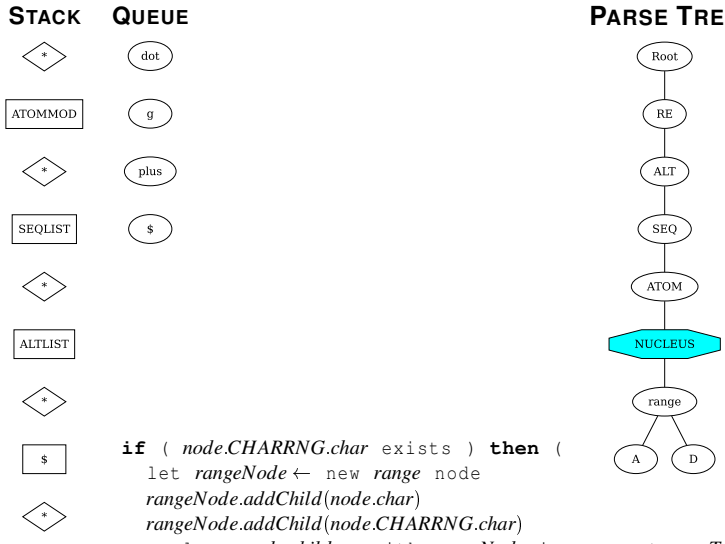
QUEUE

PARSE TREE




```
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
)
```

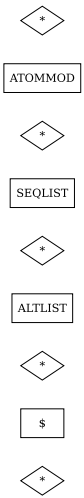
Operation: end of $NUCLEUS$ production — after rule 14 SDT procedure



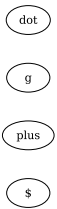

```
if ( node.CHARRNG.char exists ) then (
   let rangeNode ← new range node
   rangeNode.addChild(node.char)
   rangeNode.addChild(node.CHARRNG.char)
   replace node.children with rangeNode in parse tree T
   return
)
```

Operation: end of $NUCLEUS$ production — after rule 14 SDT procedure

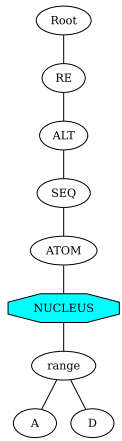
STACK



QUEUE



PARSE TREE



Notice that our tree now contains node types **not from the grammar** (range),

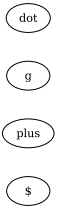
That's OK, it's your tree, you can do with it as you please :)

Input RE: **`A-D.g+`**

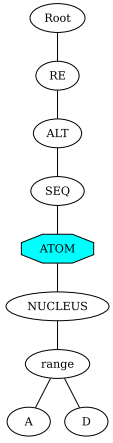
Operation: end of $NUCLEUS$ production

STACK

QUEUE



PARSE TREE

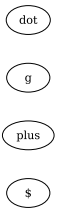


Input RE: **`A–D.g+`**

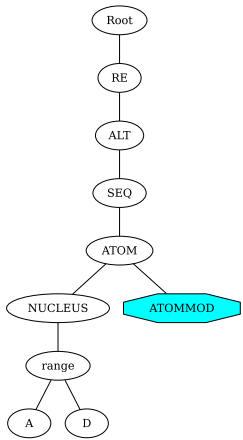
Operation: `dot` predicts rule 12 $ATOMMOD \rightarrow \lambda$

STACK

QUEUE



PARSE TREE

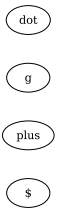


Input RE: **`A-D.g+`**

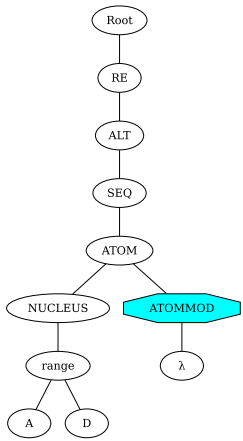
Operation: λ consumed from stack

STACK

QUEUE



PARSE TREE

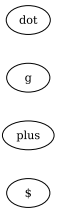


Input RE: **`A–D.g+`**

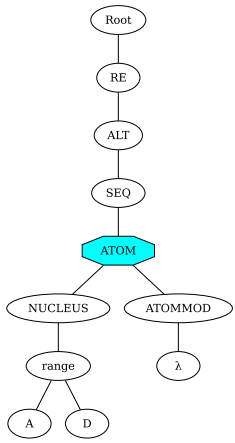
Operation: end of $ATOMMOD$ production

STACK

QUEUE



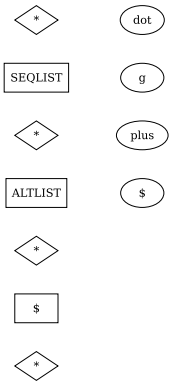
PARSE TREE



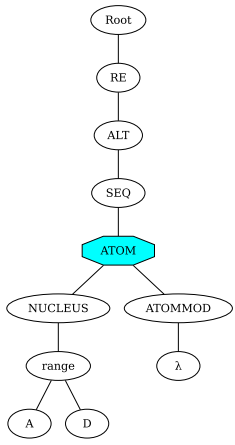
Input RE: **A–D.g+**

Operation: end of $ATOM$ production — before rule 9 SDT procedure

STACK **QUEUE**



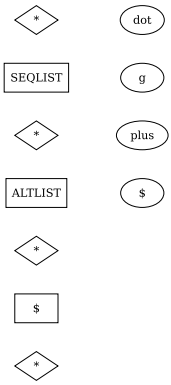
PARSE TREE



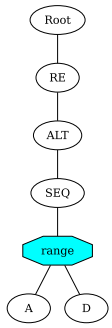

```
if ( node.ATOMMOD.child is λ ) then (
   replace node with node.children[0].child in parse tree T
   return
)
```

Operation: end of $ATOM$ production — after rule 9 SDT procedure

STACK **QUEUE**



PARSE TREE

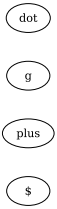



```
if ( node.ATOMMOD.child is λ ) then (
   replace node with node.children[0].child in parse tree T
   return
)
```

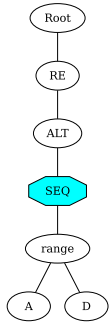
Operation: end of $ATOM$ production

STACK

QUEUE



PARSE TREE

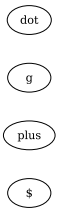


Input RE: **`A-D.g+`**

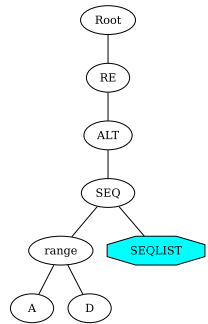
Operation: `dot` predicts rule 7 $SEQLIST \rightarrow ATOM\ SEQLIST$

STACK

QUEUE



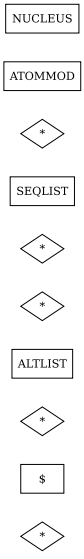
PARSE TREE



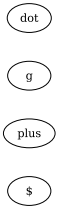
Input RE: **`A-D.g+`**

Operation: `dot` predicts rule 9 $ATOM \rightarrow NUCLEUS\ ATOMMOD$

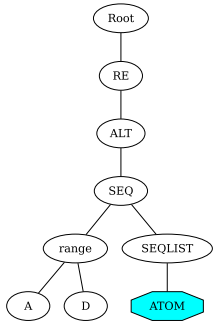
STACK



QUEUE



PARSE TREE

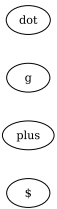


Input RE: **`A-D.g+`**

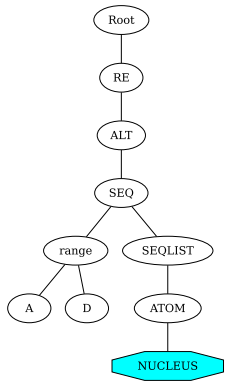
Operation: `dot` predicts rule 15 $NUCLEUS \rightarrow dot$

STACK

QUEUE



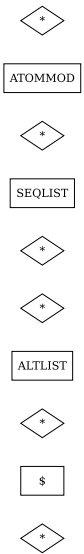
PARSE TREE



Input RE: **`A-D.g+`**

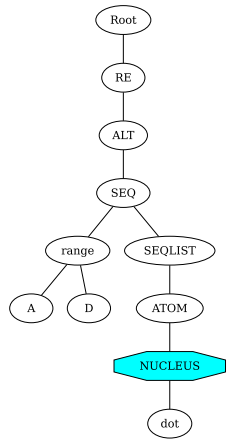
Operation: token `dot` match (`dot`)

STACK



QUEUE

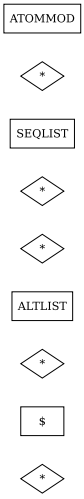
PARSE TREE



Input RE: **`A–D.g+`**

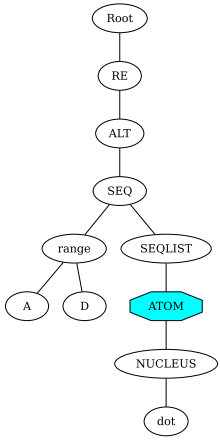
Operation: end of $NUCLEUS$ production

STACK



QUEUE

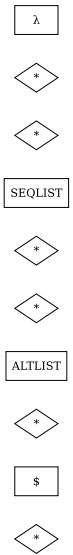
PARSE TREE



Input RE: **`A-D.g+`**

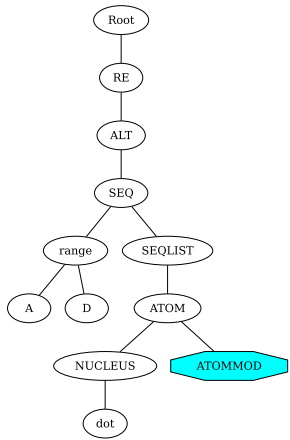
Operation: `char` predicts rule 12 $ATOMMOD \rightarrow \lambda$

STACK



QUEUE

PARSE TREE



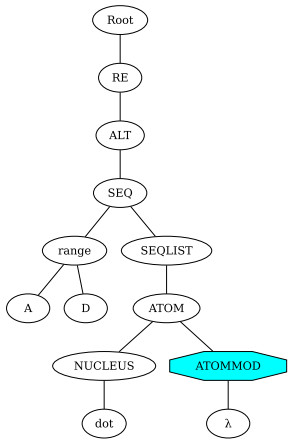
Input RE: **`A-D.g+`**

Operation: λ consumed from stack

STACK

QUEUE

PARSE TREE



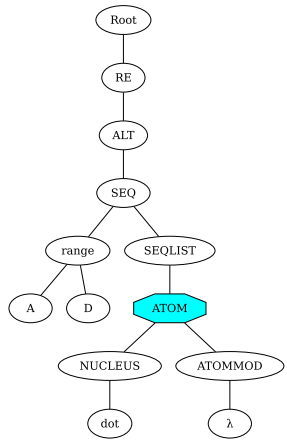
Input RE: **`A–D.g+`**

Operation: end of $ATOMMOD$ production

STACK

QUEUE

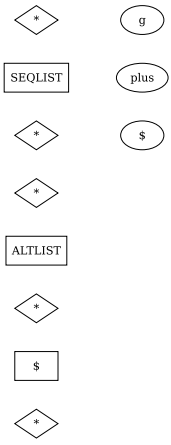
PARSE TREE



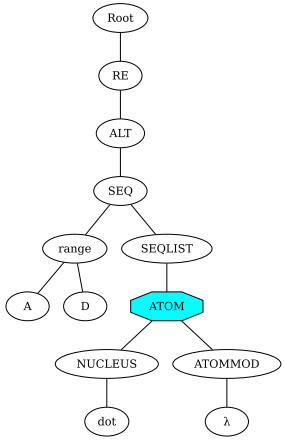
Input RE: **A–D.g+**

Operation: end of $ATOM$ production — before rule 9 SDT procedure

STACK **QUEUE**



PARSE TREE



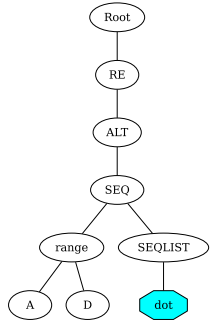

```
if ( node.ATOMMOD.child is λ ) then (
   replace node with node.children[0].child in parse tree T
   return
)
```

Operation: end of $ATOM$ production — after rule 9 SDT procedure

STACK

QUEUE

PARSE TREE



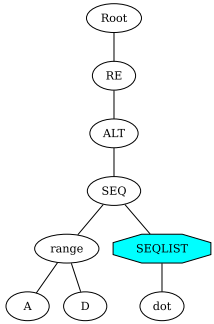

```
if ( node.ATOMMOD.child is λ ) then (
   replace node with node.children[0].child in parse tree T
   return
)
```

Operation: end of $ATOM$ production

STACK

QUEUE

PARSE TREE



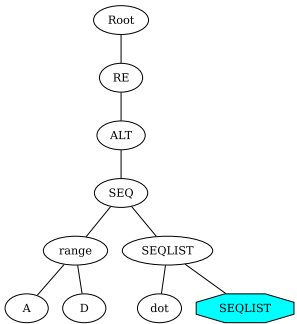
Input RE: **`A-D.g+`**

Operation: `char` predicts rule 7 $SEQLIST \rightarrow ATOM\ SEQLIST$

STACK

QUEUE

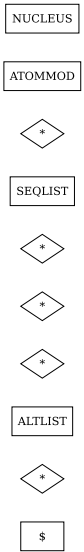
PARSE TREE



Input RE: **`A-D.g+`**

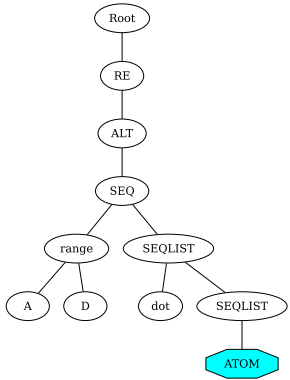
Operation: `char` predicts rule 9 $ATOM \rightarrow NUCLEUS\,ATOMMOD$

STACK



QUEUE

PARSE TREE



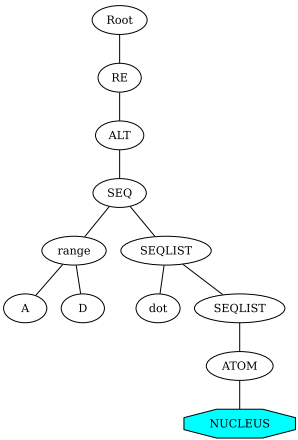
Input RE: **`A-D.g+`**

Operation: `char` predicts rule 14 $NUCLEUS \rightarrow char\ CHARRNG$

STACK

QUEUE

PARSE TREE



Input RE: **`A-D.g+`**

Operation: token `char` match (`char`)

STACK

CHARRNG

*

ATOMMOD

*

SEQLIST

*

*

*

ALTLIST

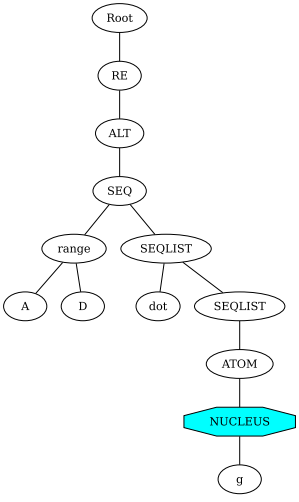
*

QUEUE

plus

$

PARSE TREE



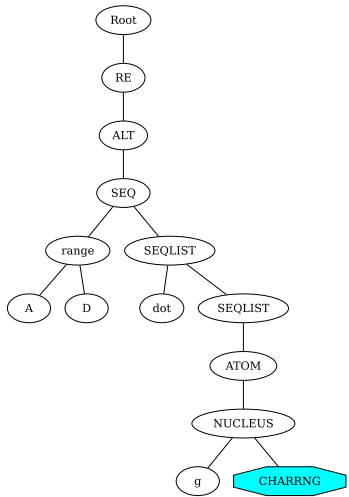
Input RE: **`A–D.g+`**

Operation: `plus` predicts rule 17 $CHARRNG \rightarrow \lambda$

STACK

QUEUE

PARSE TREE



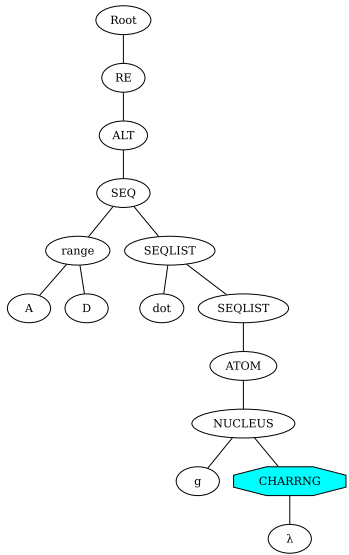
Input RE: **`A-D.g+`**

Operation: λ consumed from stack

STACK

QUEUE

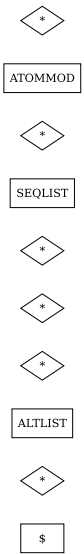
PARSE TREE



Input RE: **`A-D.g+`**

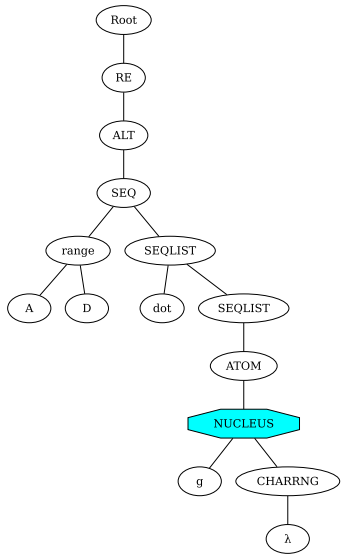
Operation: end of *CHARRNG* production

STACK



QUEUE

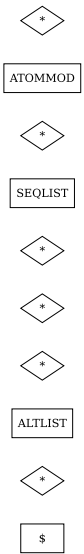
PARSE TREE



Input RE: **`A–D.g+`**

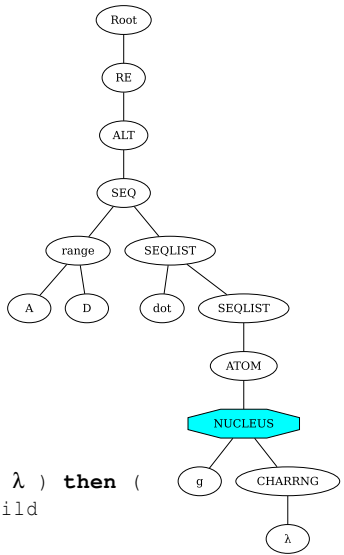
Operation: end of $NUCLEUS$ production — before rule 14 SDT procedure

STACK



QUEUE

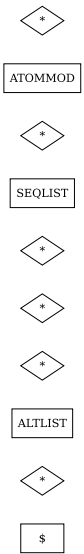
PARSE TREE




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

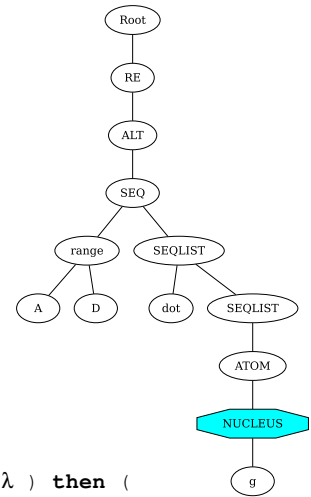
Operation: end of $NUCLEUS$ production — after rule 14 SDT procedure

STACK



QUEUE

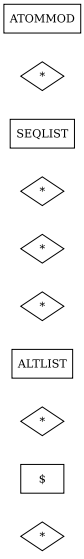
PARSE TREE




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

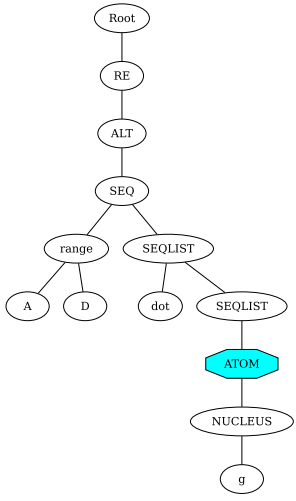
Operation: end of $NUCLEUS$ production

STACK



QUEUE

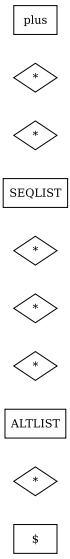
PARSE TREE



Input RE: **`A–D.g+`**

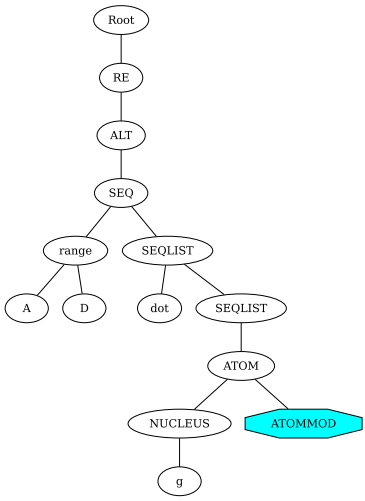
Operation: `plus` predicts rule 11 $ATOMMOD \rightarrow plus$

STACK



QUEUE

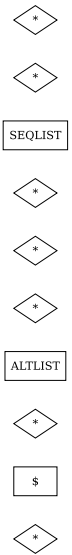
PARSE TREE



Input RE: **`A-D.g+`**

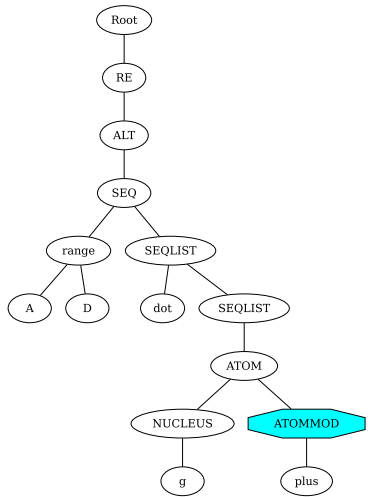
Operation: token `plus` match (`plus`)

STACK



QUEUE

PARSE TREE



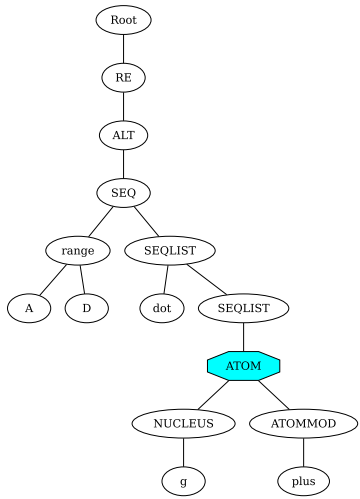
Input RE: **`A-D.g+`**

Operation: end of $ATOMMOD$ production

STACK

QUEUE

PARSE TREE

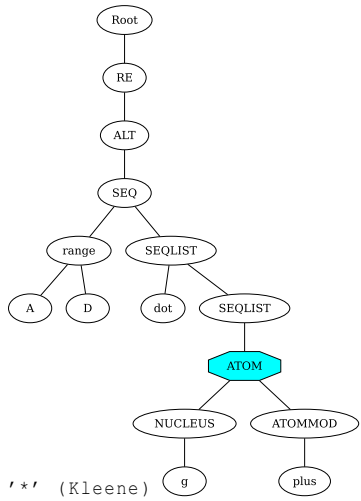


Input RE: **`A-D.g+`**

Operation: end of $ATOM$ production — before rule 9 SDT procedure

STACK **QUEUE**

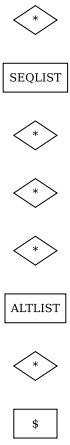
PARSE TREE




```
# ATOMMOD.child is either '+' or '*' (Kleene)
let newAtom ← new node.ATOMMOD.child node
newAtom.addChild( node.children[0].child )
replace node with newAtom in parse tree T
```

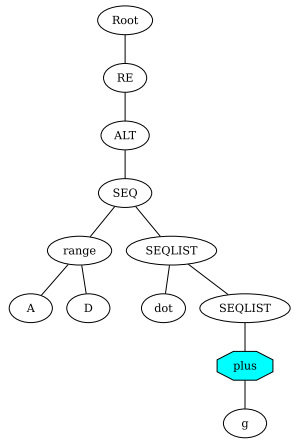
Operation: end of $ATOM$ production — after rule 9 SDT procedure

STACK



QUEUE

PARSE TREE



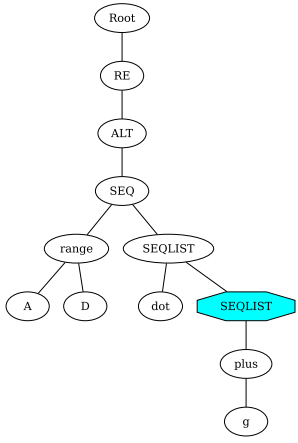

```
# ATOMMOD.child is either '+' or '*' (Kleene)
let newAtom ← new node.ATOMMOD.child node
newAtom.addChild( node.children[0].child )
replace node with newAtom in parse tree T
```

Operation: end of $ATOM$ production

STACK

QUEUE

PARSE TREE



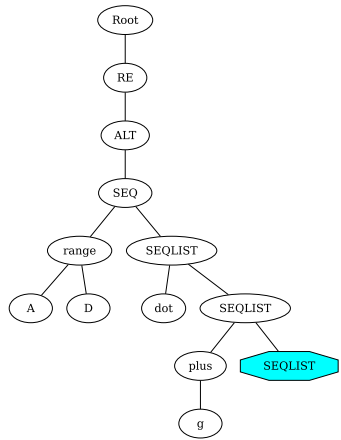
Input RE: **A–D.g+**

Operation: $ predicts rule 8 $SEQLIST \rightarrow \lambda$

STACK

QUEUE

PARSE TREE



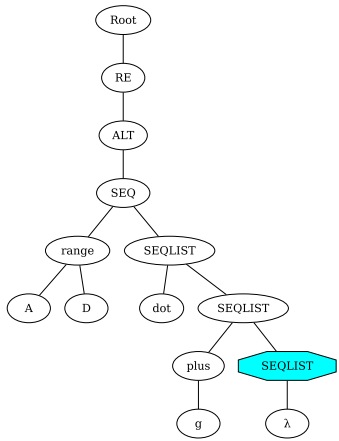
Input RE: **A–D.g+**

Operation: λ consumed from stack

STACK

QUEUE

PARSE TREE



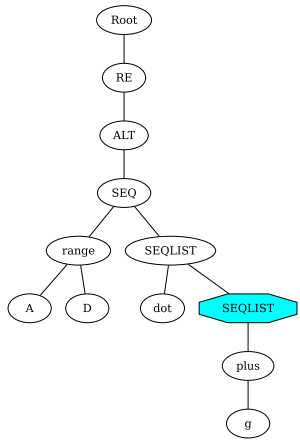
Input RE: **`A–D.g+`**

Operation: end of $SEQLIST$ production

STACK

QUEUE

PARSE TREE



Whoa! There was an SDT procedure in there. . . What was its logic?

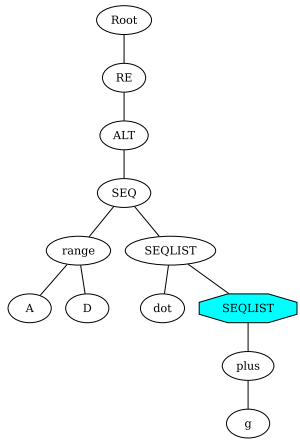
Input RE: **`A-D.g+`**

Operation: end of $SEQLIST$ production

STACK

QUEUE

PARSE TREE



Whoa! There was an SDT procedure in there... What was its logic?

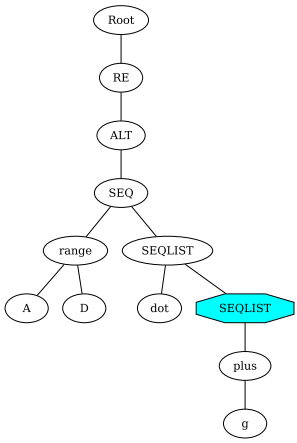
`remove parent's right most child`

Input RE: **`A-D.g+`**

Operation: end of $SEQLIST$ production

STACK **QUEUE**

PARSE TREE

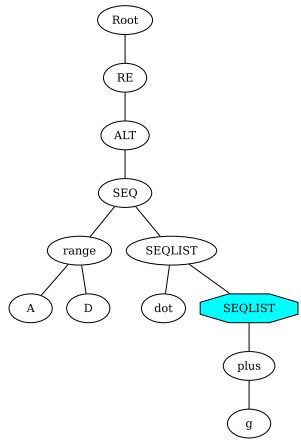


There is an implementation nuance here, are the SDT procedures expected to manage the notion of **the current node** of the parsing engine? In this particular case, the procedure could make the (new) right most node of *node.parent* the current node (**`plus`**); alternatively a **current node stack** could be used instead of a singular notion of the current node.

Operation: end of $SEQLIST$ production

STACK **QUEUE**

PARSE TREE



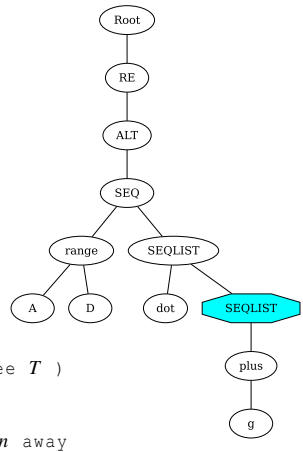
Yet **another approach** to avoiding *current node corruption* is to permit SDT procedures to manipulate **only their own and their descendent's structures**. This "don't talk back to your parent" approach works particularly well for SDT procedures in LR parses. Avoid mixing and matching your SDT philosophies within one grammar and implementation.

Operation: end of $SEQLIST$ production — before rule 7 SDT procedure

STACK

QUEUE

PARSE TREE



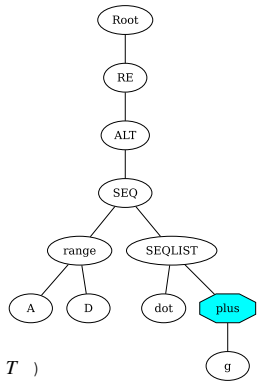

```
procedure SEQLIST( node, parse tree T )
  let parent ← parent of node in T
  if ( node.child is λ ) then (
    trim last child of parent.children away
  ) else if ( parent is SEQLIST ) then (
    let myChildren = node.children
    trim last child of parent.children away
    append parent.children with myChildren
  )
```

Operation: end of $SEQLIST$ production — after rule 7 SDT procedure

STACK

QUEUE

PARSE TREE



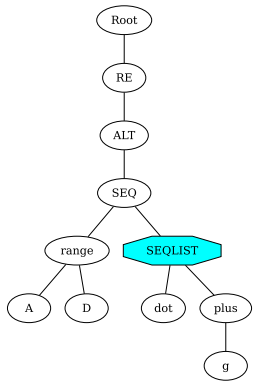

```
procedure SEQLIST( node, parse tree T )
  let parent ← parent of node in T
  if ( node.child is λ ) then (
    trim last child of parent.children away
  ) else if ( parent is SEQLIST ) then (
    let myChildren = node.children
    trim last child of parent.children away
    append parent.children with myChildren
  )
```

Operation: end of $SEQLIST$ production

STACK

QUEUE

PARSE TREE



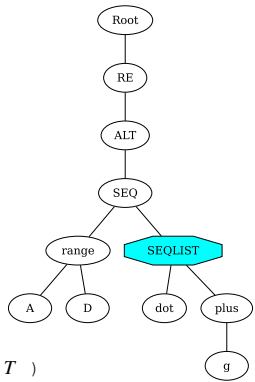
Input RE: **`A-D.g+`**

Operation: end of *SEQLIST* production — before rule 7 SDT procedure

STACK

QUEUE

PARSE TREE



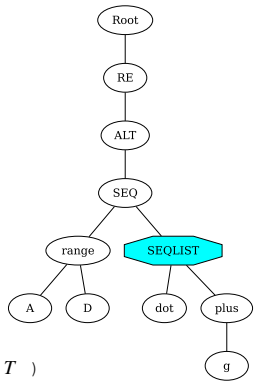

```
procedure SEQLIST( node, parse tree T )
  let parent ← parent of node in T
  if ( node.child is λ ) then (
    trim last child of parent.children away
  ) else if ( parent is SEQLIST ) then (
    let myChildren = node.children
    trim last child of parent.children away
    append parent.children with myChildren
  )
```

Operation: end of *SEQLIST* production — after rule 7 SDT procedure

STACK

QUEUE

PARSE TREE



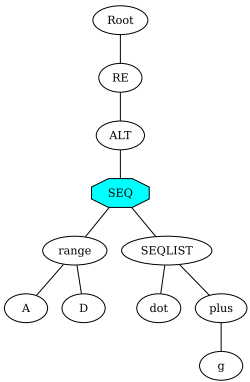

```
procedure SEQLIST( node, parse tree T )
  let parent ← parent of node in T
  if ( node.child is λ ) then (
    trim last child of parent.children away
  ) else if ( parent is SEQLIST ) then (
    let myChildren = node.children
    trim last child of parent.children away
    append parent.children with myChildren
  )
```

Operation: end of $SEQLIST$ production

STACK

QUEUE

PARSE TREE

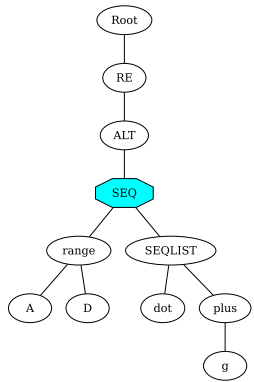


Input RE: **`A-D.g+`**

Operation: end of SEQ production — before rule 5 SDT procedure

STACK **QUEUE**

PARSE TREE



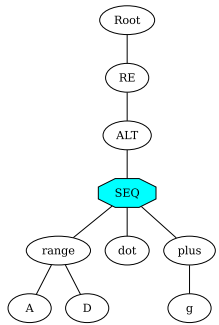

```
procedure SEQ( node, parse tree T )
  if ( node.children.SEQLIST exists ) then (
    replace node.children.SEQLIST with node.children.SEQLIST's children
  )
  if ( |node.children| = 1 ) then (
    replace node with node.child in parse tree T
  )
```

Operation: end of SEQ production — after rule 5 SDT procedure

STACK

QUEUE

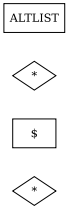
PARSE TREE




```
procedure SEQ( node, parse tree T )
  if ( node.children.SEQLIST exists ) then (
    replace node.children.SEQLIST with node.children.SEQLIST's children
  )
  if ( |node.children| = 1 ) then (
    replace node with node.child in parse tree T
  )
```

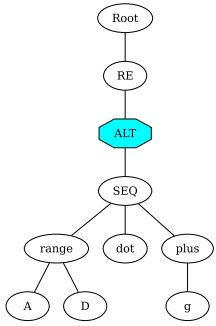
Operation: end of SEQ production

STACK



QUEUE

PARSE TREE



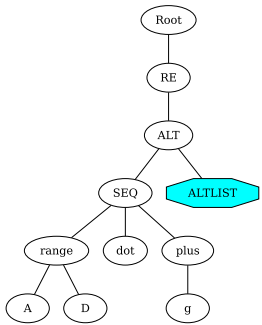
Input RE: **`A-D.g+`**

Operation: $ predicts rule 4 $ALTLIST \rightarrow \lambda$

STACK

QUEUE

PARSE TREE

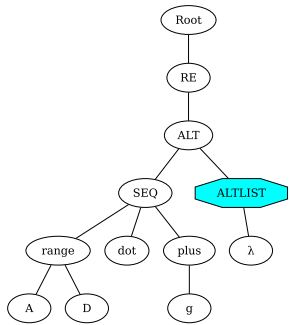


Input RE: **A–D.g+**

Operation: λ consumed from stack

STACK **QUEUE**

PARSE TREE

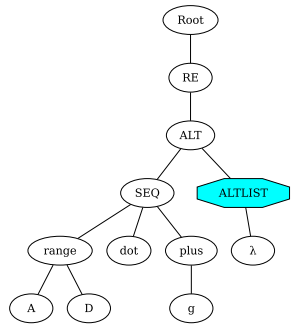


Input RE: **A–D.g+**

Operation: end of $ALTLIST$ production — before rule 4 SDT procedure

STACK **QUEUE**

PARSE TREE

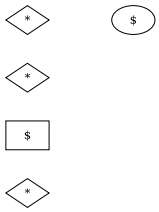


What will the SDT procedure for $ALTLIST \rightarrow \lambda$ production do?

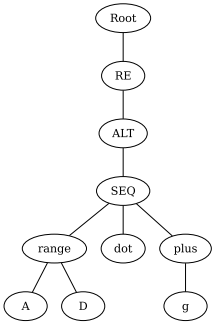
Input RE: **`A-D.g+`**

Operation: end of $ALTLIST$ production — after rule 4 SDT procedure

STACK **QUEUE**



PARSE TREE



Remove its parent's right most child.

Here is another instance where the *current node* notion in the parsing algorithm can fail if not coded carefully (because the current node was just lobbed off the tree).

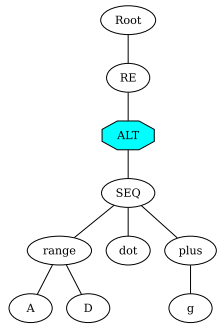
Input RE: **`A-D.g+`**

Operation: end of $ALTLIST$ production

STACK

QUEUE

PARSE TREE



What should $ALT \rightarrow SEQ\,ALTLIST$ production do with only one child?

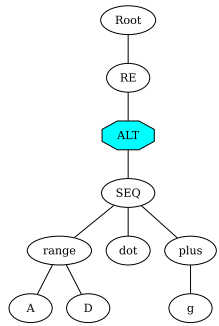
Input RE: **`A-D.g+`**

Operation: end of $ALT\,LIST$ production

STACK

QUEUE

PARSE TREE



What should $ALT \rightarrow SEQ\,ALT\,LIST$ production do with only one child?

Replace itself with its child.

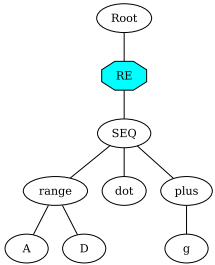
Input RE: **`A-D.g+`**

Operation: end of ALT production

STACK

QUEUE

PARSE TREE

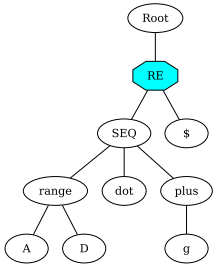


Input RE: **`A-D.g+`**

Operation: token $ match ($)

STACK **QUEUE**

PARSE TREE



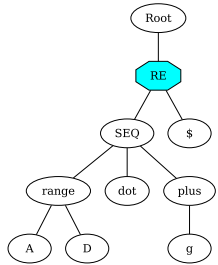
And what about the $RE \rightarrow ALT\ \$$ production?

Input RE: **`A-D.g+`**

Operation: token $ match ($)

STACK **QUEUE**

PARSE TREE



And what about the $RE \rightarrow ALT\ \$$ production? Replace itself with its left most child.

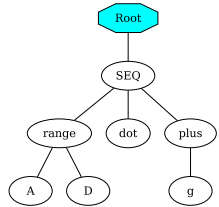
Beware, the farther up the parse tree you progress,
the less likely that a node's children will match a production rule's RHS!

Input RE: **`A-D.g+`**

Operation: end of RE production

STACK **QUEUE**

PARSE TREE



Voilà, a proper RE expression tree. Your LGA tonight will ask you to complete the pseudo code for the other production rules in our RE grammar.