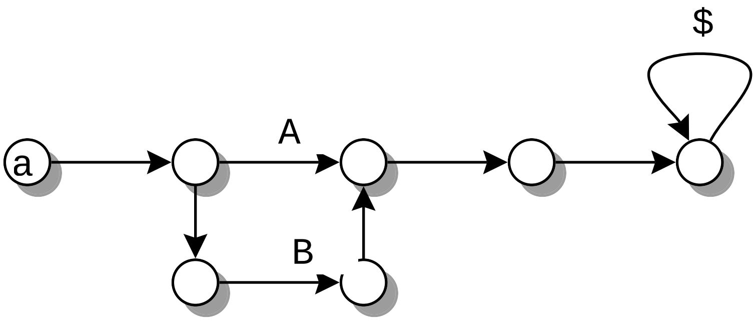
How To Compute LL(k) Lookahead

- GLA: Grammar Lookahead Automata
  NFA that encodes the set of all possible lookahead strings for any parsing strategy.
- Lookahead computation
  Like NFA->DFA conversion; here, we do bounded walk of the GLA.

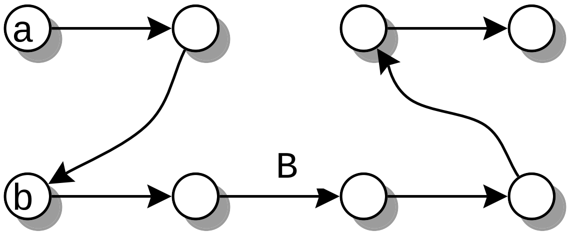
Alternative productions (and EOF loop).

```
a   :   A       // { A }
    |   B       // { B }
    ;
```



Rule References

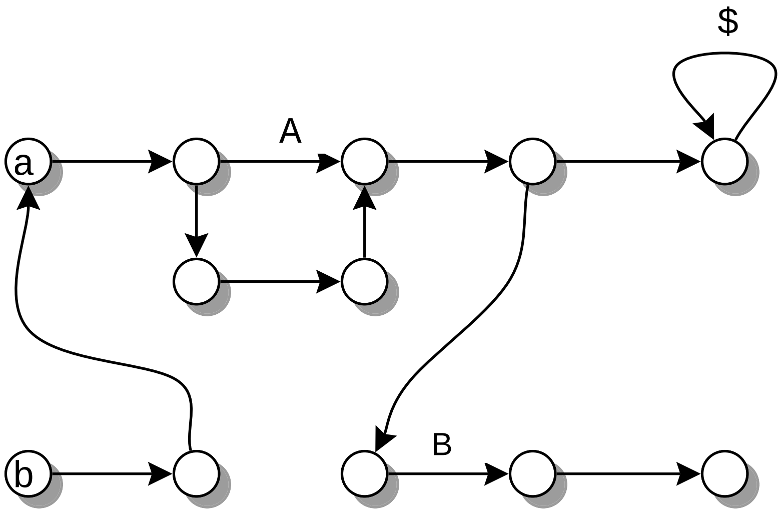
```
a   :   b ;     // { B }
b   :   B ;     // { B }
```



Epsilon transfers and rule references

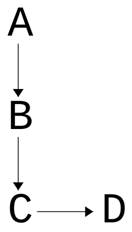
```
a   :   A        // { A }
    |            // { $, B }
    ;

b   :   a B      // { A, B }
    ;
```

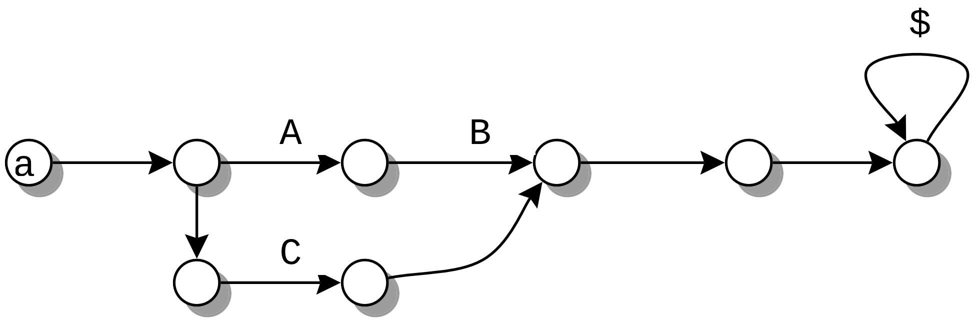


LL(k) for k>1!!!!

- Store lookahead strings in a tree of depth k
- 3-strings “`A B C`” and “`A B D`” encoded as



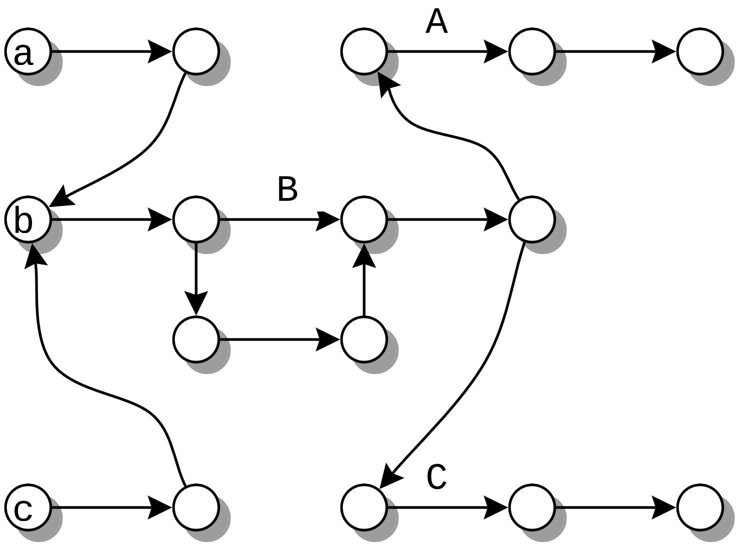
```
a   :   A B      // { AB }
    |   C        // { C$ }
    ;
```



```
a   :   b A     // { BA, A$ }
    ;

b   :   B       // { BA, BC }
    |           // { A$, C$ }
    ;

c   :   b C     // { BC, C$ }
    ;
```



LL(k) Lookahead Computation Algorithm

```
function $LOOK_k$( $p$ : $Node$ ) returns tree of $terminal$;
begin
    var $t, u$ : tree of $terminal$;

    if $p$=nil or $k$=0 then return nil;
    if $p.busy[k]$ then return nil;
    $p.busy[k]$ = true;

    if ( $p.edge_1$ is-a-terminal )
    begin
        $q = p.label_1$;
        $r = LOOK_{k-1}$( $p.edge_1$ );
        $t = \begin{array}{c} q \\ \downarrow \\ r \end{array}$ ;
    end
    else
        $t = LOOK_k$ ( $p.edge_1$ );
    $u = LOOK_k$ ( $p.edge_2$ );
    $p.busy[k]$ = false;
    if $t$=nil then return $u$;
    else return $t \longrightarrow u$ ;
end $LOOK$ sub $k$;
```