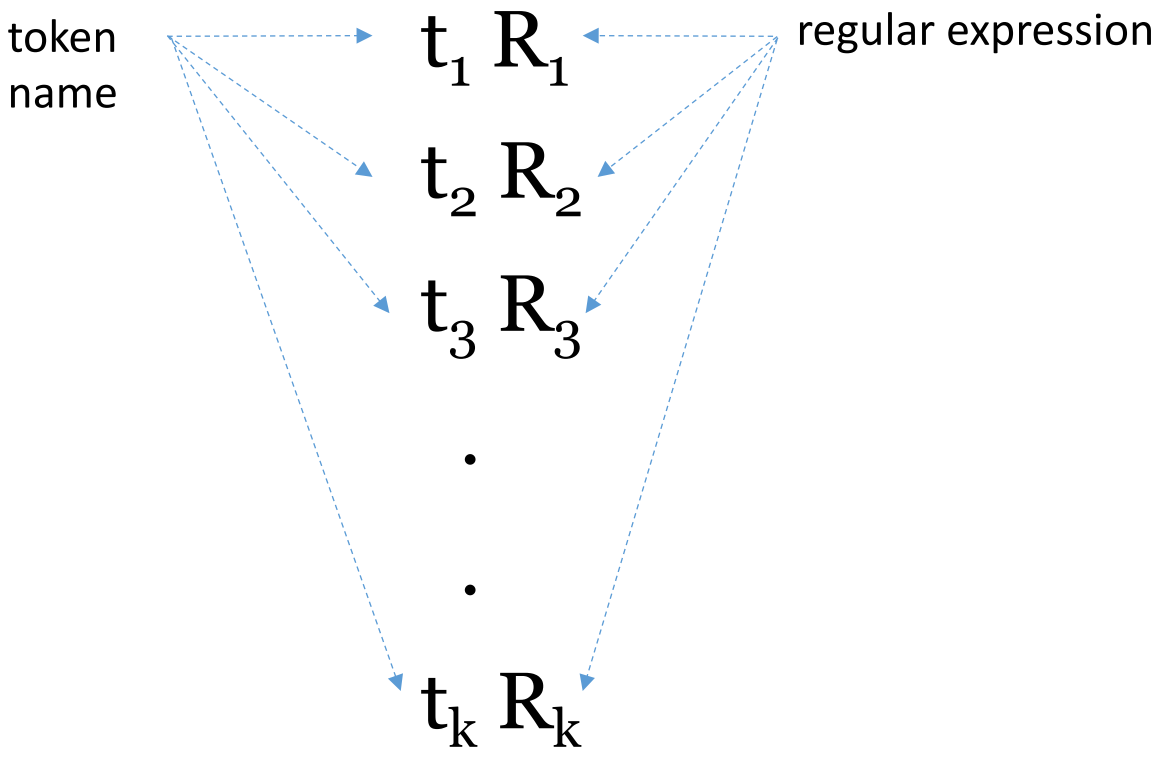
Implementing my_GetToken() Automatically

Rida Bazzi

CSE 340 SPRING 2021

This is not meant to replace the project description!

Generating My_GetToken() automatically



Generating My_GetToken() automatically

$$\left.\begin{array}{l} t_1\ R_1 \\ t_2\ R_2 \\ t_3\ R_3 \\ \quad . \\ \quad . \\ t_k\ R_k \end{array}\right\} \Rightarrow \text{My\_GetToken()}$$

Generating My_GetToken() automatically

$$\left.\begin{array}{l} t_1\ R_1 \\ t_2\ R_2 \\ t_3\ R_3 \\ \quad . \\ \quad . \\ t_k\ R_k \end{array}\right\} \Rightarrow \text{My\_GetToken()}$$

Implementing my_GetToken() automatically

- Given a list of token names and regular expressions, one for each token, implement my_GetToken() function

- The function should correctly implement
  - longest matching prefix rule (we have seen this in the first week)
  - breaking ties according to list order (we have seen this also)

- We can assume that none of the regular expressions given for the tokens has epsilon in its language, which is to be expected, because epsilon is not a token!

Note: the language of a regular expression is the set of strings that the expression represents. See project document for more details.

function match()

Function

match(REG *r*, string *s*, int *p*)

Input is

- *r* REG
- *s* is a string
- *p* is a position in string s

Behavior

1. determine the longest possible substring of s, starting at position p, that matches the regular expression represented by r
2. If there is no match, that will be indicated.
3. Note that match need not return an actual substring, it just needs to return a position p’ corresponding to the end of the substring. The substring would be between p and p’

function my_getToken()

Function

my_GetToken(Token_List *L*, string *s*, int *p*)

Input

- *L* is a list of tokens, where each entry in the list consists of a token name and a REG
- *s* is a string
- *p* is a position in string s

Behavior

1. call match(*r*,*s*,*p*) for each REG in the list *L*
2. For each REG *r*, records the longest matching prefix obtained from the call match(*r*,*s*,*p*)
3. Returns the token for which match(*r*,*s*,*p*) returns the longest amongst all the prefixes obtained in step 2 and advance the position to reflect that the input is consumed
4. If there is a tie, return the token listed first in the list

Plan

- I will explain how to construct REGs
- I will then explain how to implement the function match
- Then I will explain how to implement my_getToken()

Constructing REGs

- REGs will be constructed recursively as shown on the following pages.

- Each REG is a directed graph

- Each REG has two special nodes
  - a starting node
  - an accepting node

- Labels on edges are characters of the alphabet or epsilon (for which we will use _ (underscore) as the label

Approach

1. Transform each expression into a graph that I will call REG (Regular Expression Graph)

2. Write a function match(REG *r*, string *s*, int *p*) that, given

    1. a REG *r*
    2. a string *s*
    3. a position *p* in the string *s*,

    returns the longest matching substring in *s*, starting at position *p*, that is in the language of the expression of REG r

Approach (cont'd)

3. Write a function GetToken(Token_List *L*, string *s*, int *p*)

Approach (cont'd)

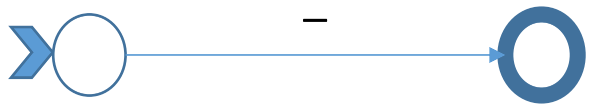
3. Write a function my_GetToken(Token_List *L*, string *s*, int *p*) that, given

    - *L* is a list of tokens, where each entry in the list consists of a token name and a REG
    - *s* is a string
    - *p* is a position in string s

   will

    1. call match(*r*,*s*,*p*) for each REG in the list *L*
    2. For each REG *r*, records the longest matching prefix obtained from the call match(*r*,*s*,*p*)
    3. Returns the token for which match(*r*,*s*,*p*) returns the longest amongst all the prefixes obtained in step 2 and advance the position to reflect that the input is consumed
    4. If there is a tie, return the token listed first in the list

REG for _

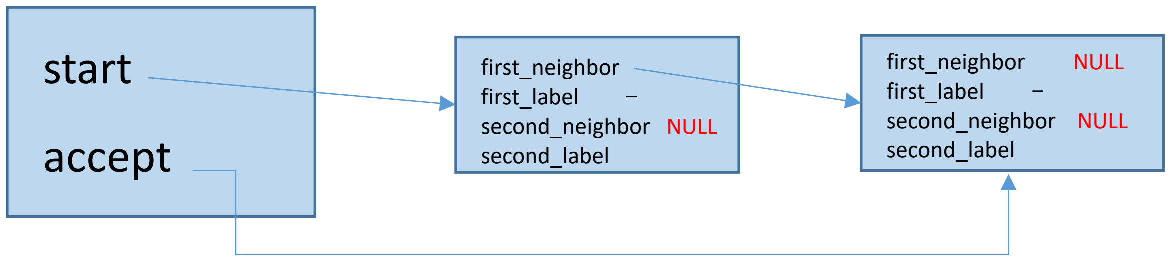


REG for _

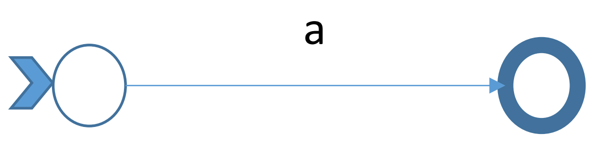
Notation

accept node
start node

If the regular expression is _ , the REG can be constructed immediately. The resulting REG is illustrated above as a graph and below as a data structure

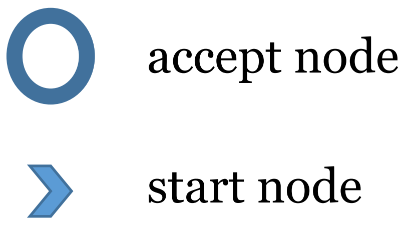


REG for a

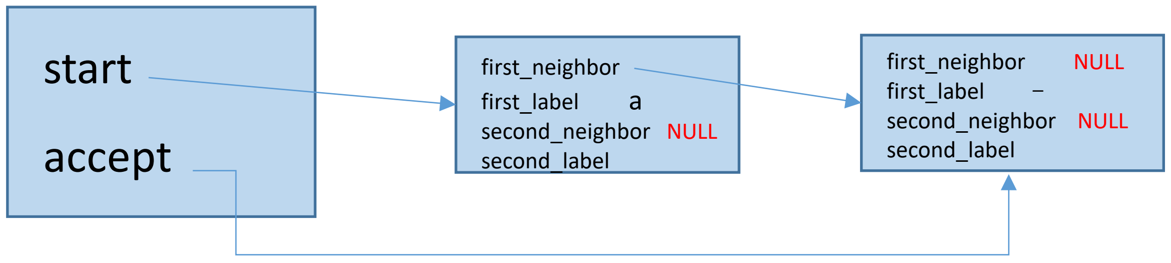


REG for a

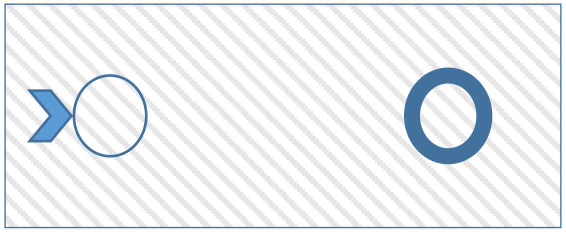
Notation



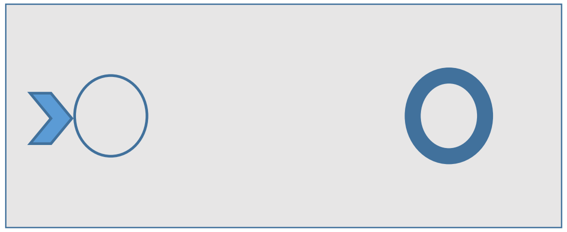
If the regular expression is a , where a is a character of the alphabet or a digit, the REG can also be constructed immediately. The resulting REG is illustrated above as a graph and below as a data structure



REG for $(R_1).(R_2)$

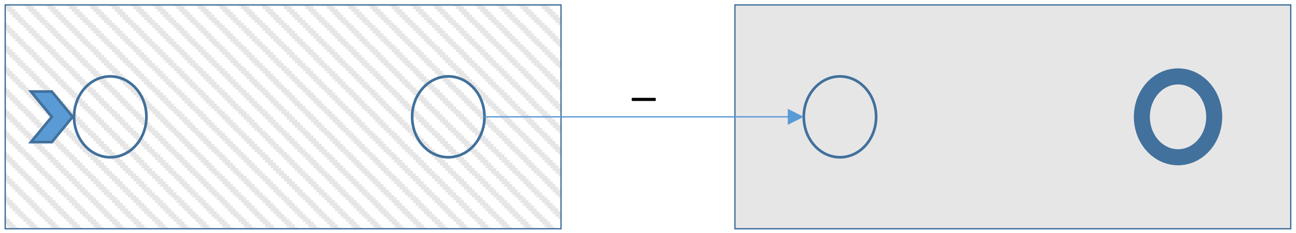


REG for R_1



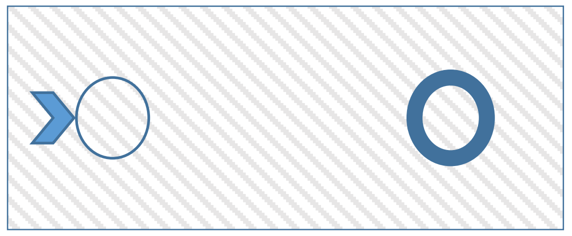
REG for R_2

If we have two expressions R_1 and R_2, we can construct the REG for $(R_1).(R_2)$ from the REGs of R_1 and R_2 as shown below.

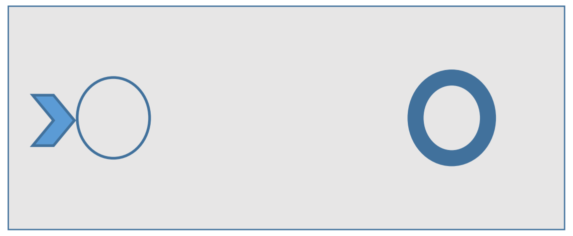


REG for $(R_1).(R_2)$

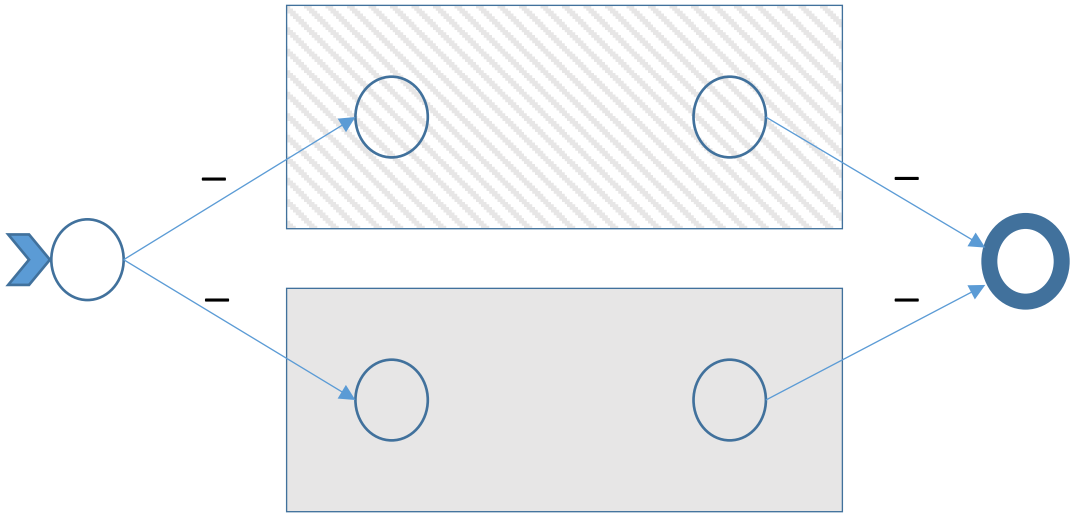
REG for $(R_1)|(R_2)$



REG for R_1

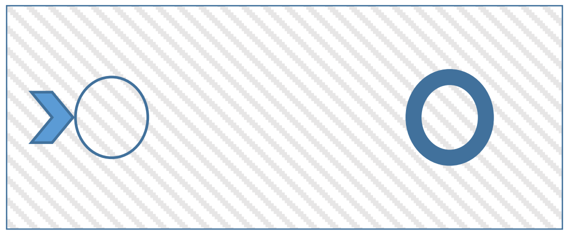


REG for R_2

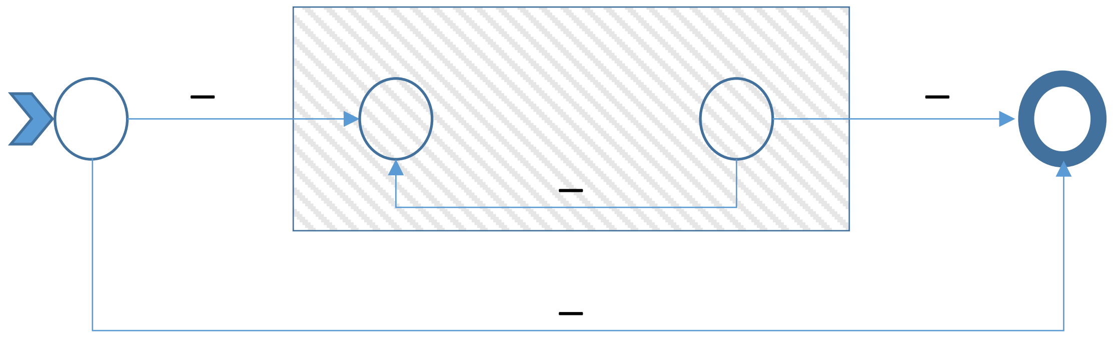


REG for $(R_1)|(R_2)$

REG for (R)*



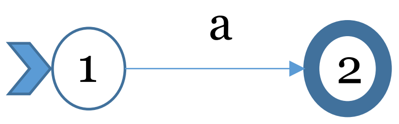
REG for R



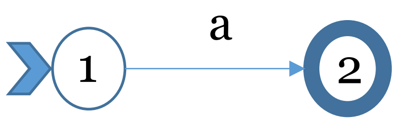
REG for (R)*

Examples

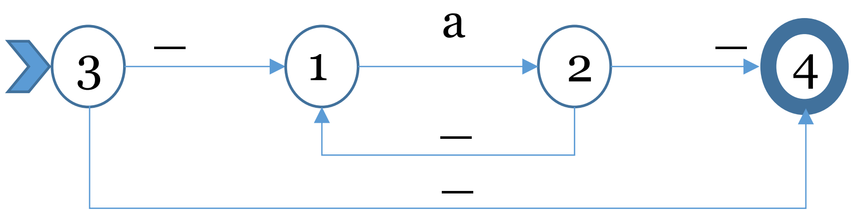
- In what follows we show how the construction works for a couple examples
- I will first show the graph illustration then I will show how the data structures look



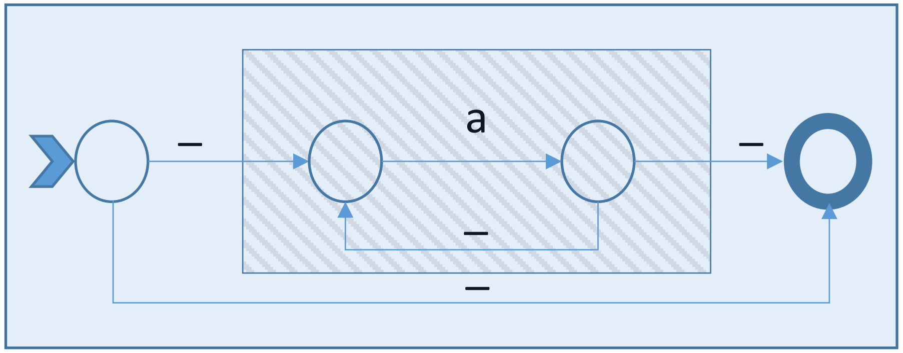
REG for a



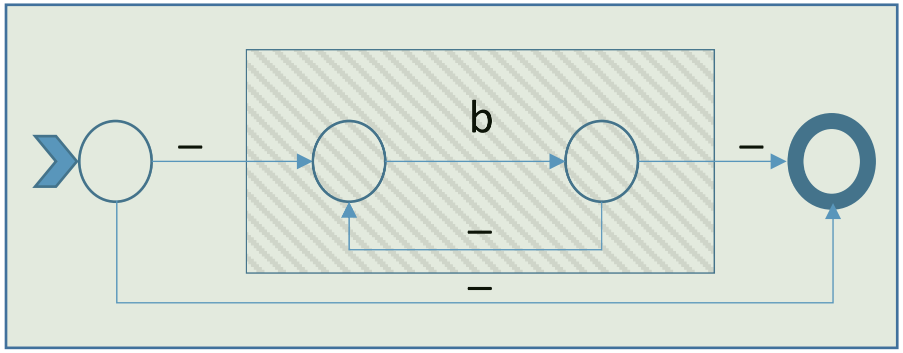
REG for a



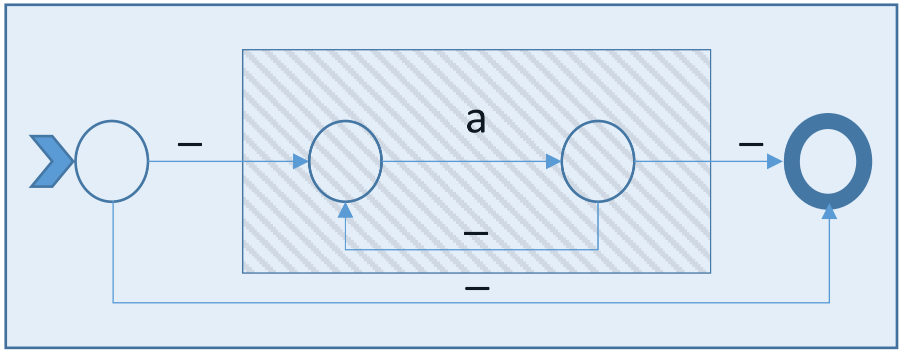
REG for (a)*



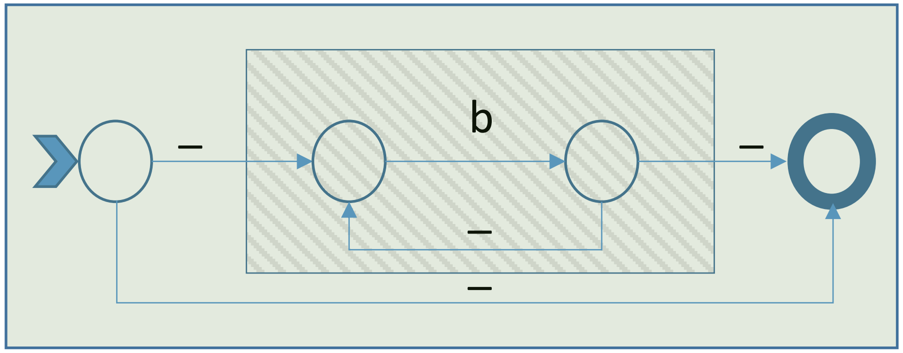
REG for (a)*



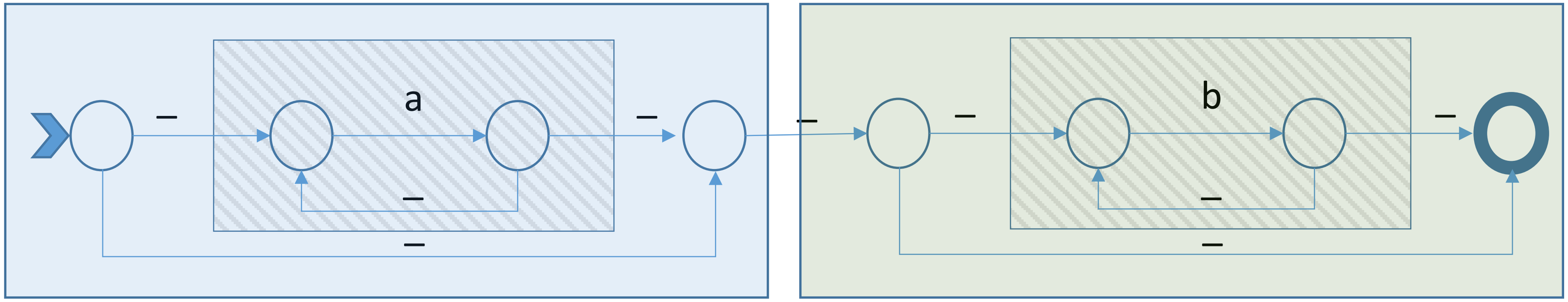
REG for (b)*



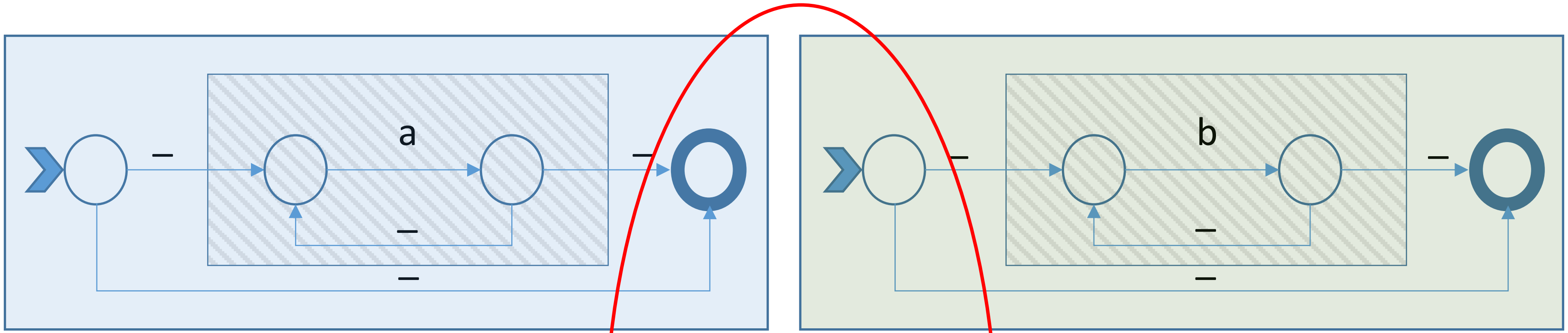
REG for (a)*



REG for (b)*

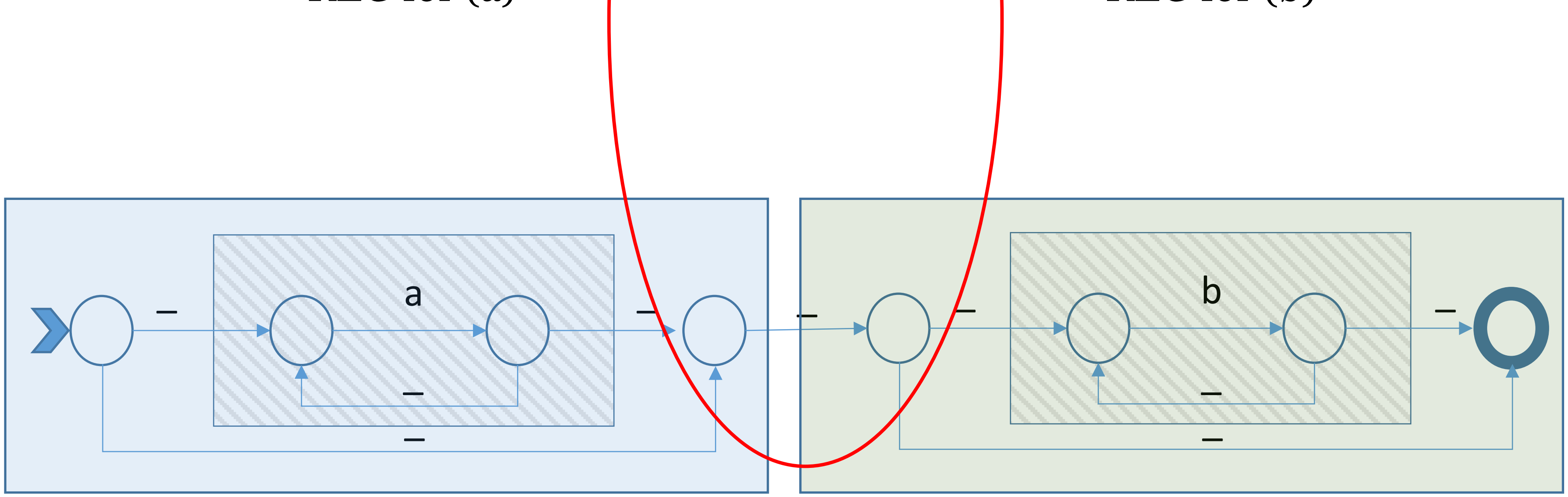


REG for ((a)*).((b)*)

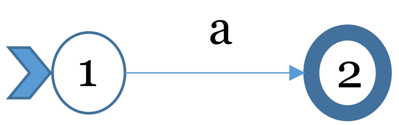


REG for (a)*

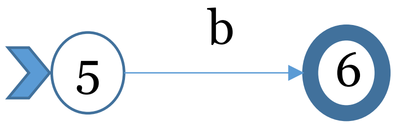
REG for (b)*



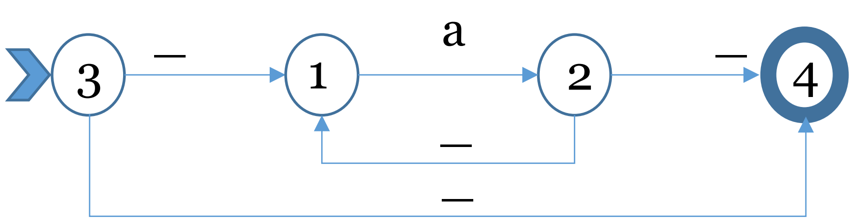
REG for ((a)*).((b)*)



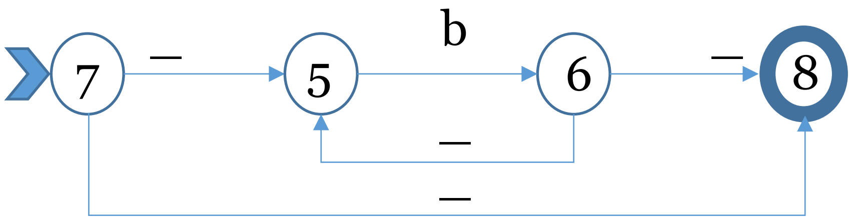
REG for a



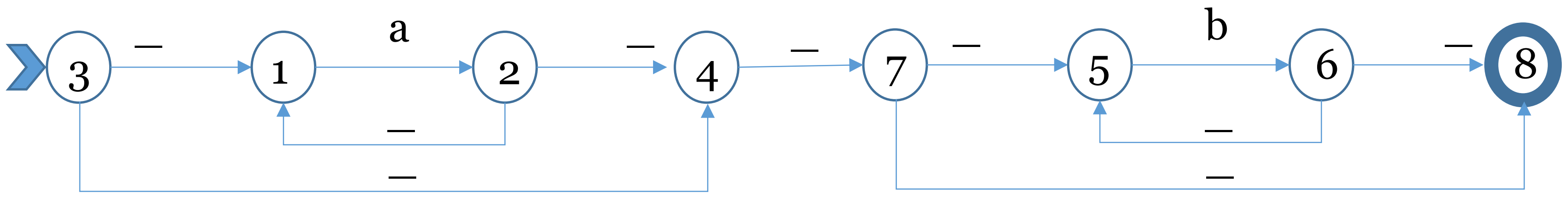
REG for b



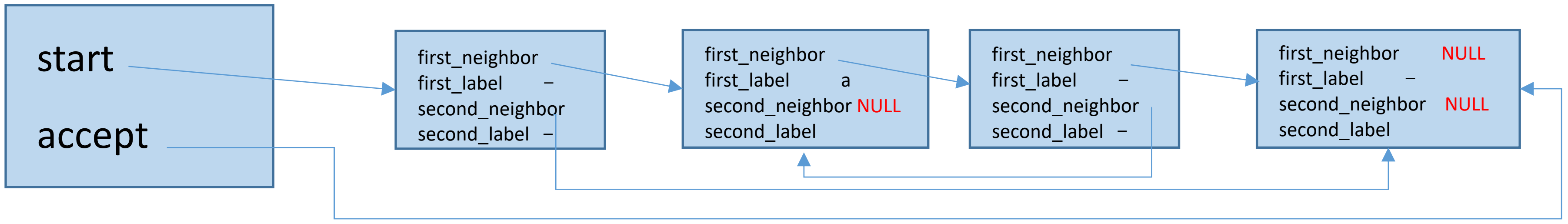
REG for (a)*



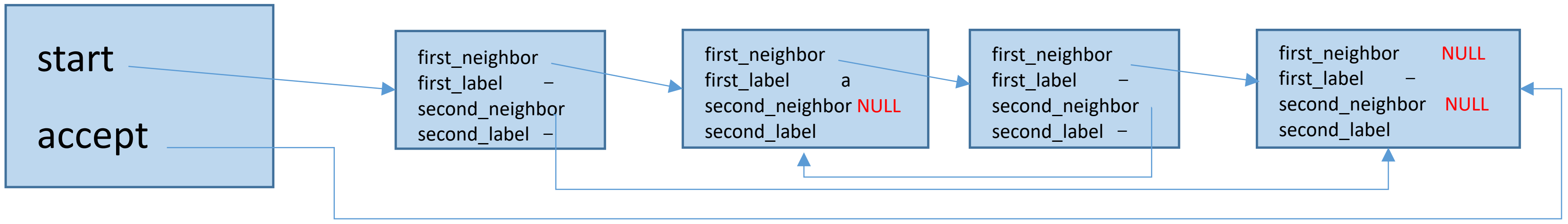
REG for (b)*



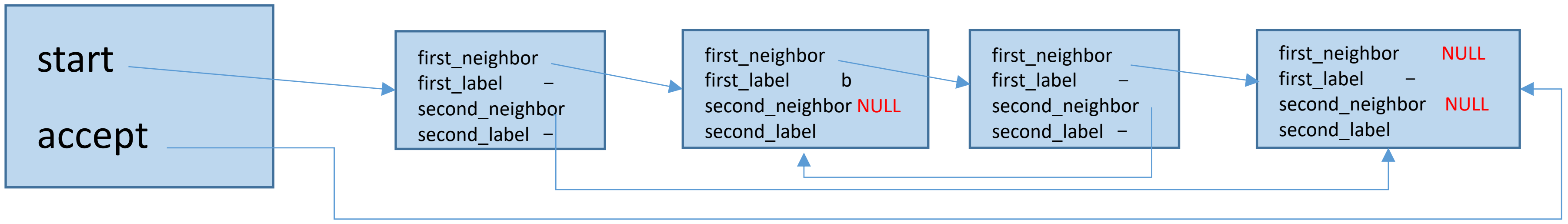
REG for ((a)*).((b)*)



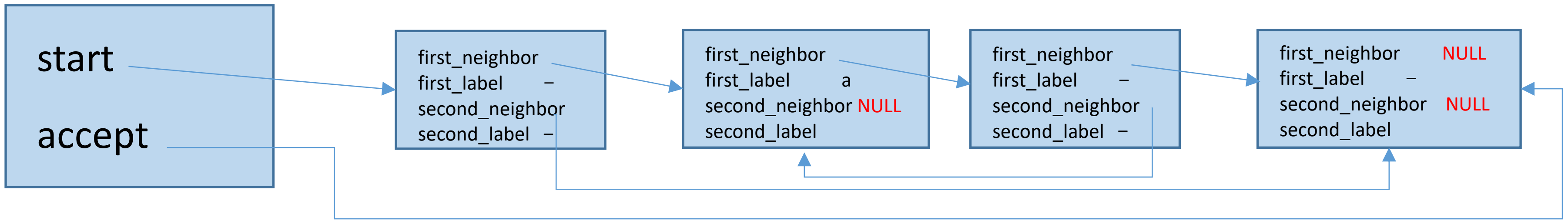
REG Data Structure for (a)*



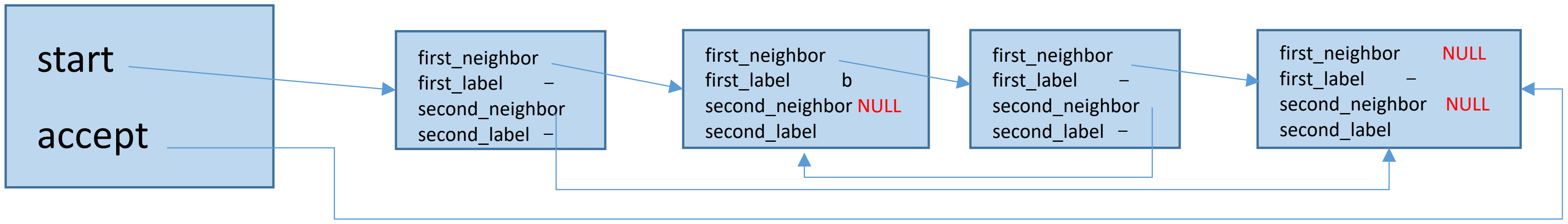
REG Data Structure for (a)*



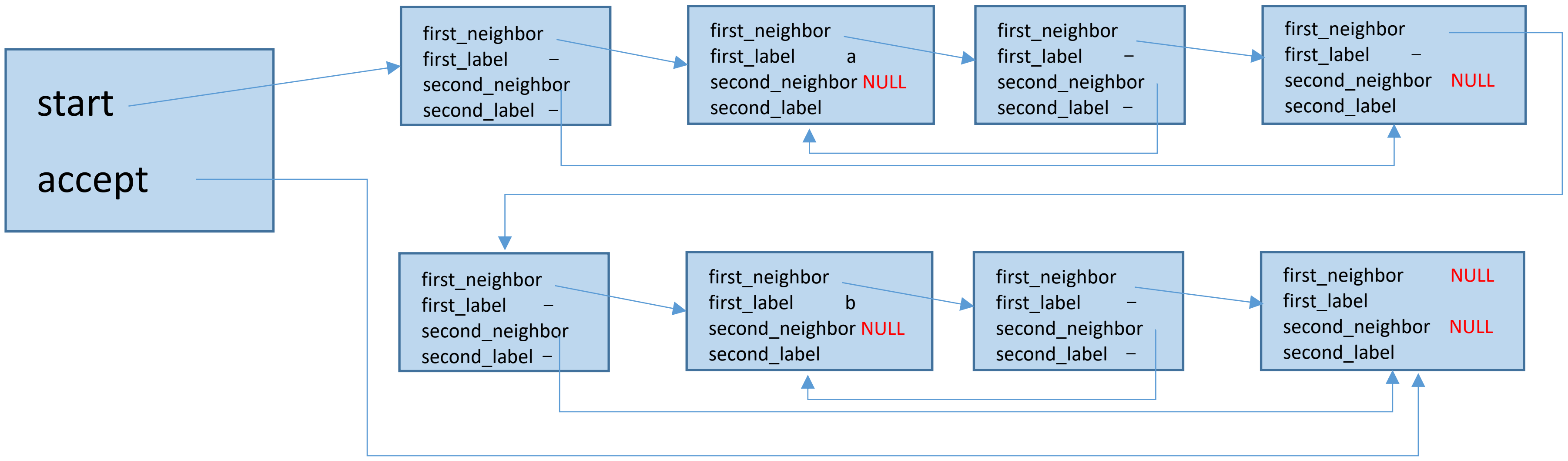
REG Data Structure for (b)*



REG Data Structure for (a)*



REG Data Structure for (b)*



REG Data Structure for ((a)*).((b)*)

Another example

- In what follows, I will assume that parse_expr() returns REGs as discussed above
- I will show a step by step execution parse_expr() on the expression ((a)*).((b)*)

parse_expr()

Input ((a) *) . ((b) *)

parse_expr()

consume LPAREN

$\boxed{(}$ (a) *) . ((b) *)

parse_expr()

$\boxed{(}$ (a) *) . ((b) *)

consume LPAREN
R1 = **parse_expr()**

parse_expr()

$\boxed{(\quad (} \quad a \quad) \quad * \quad) \; . \; (\quad (\quad b \quad) \quad * \quad)$

```
consume LPAREN
R1 = parse_expr()
        consume LPAREN
```

$\boxed{(\quad (}\quad a\quad)\quad *\quad)\ .\ (\quad (\quad b\quad)\quad *\quad)$

```
parse_expr()
            consume LPAREN
            R1 = parse_expr()
                        consume LPAREN
                        R1  = parse_expr()
```

parse_expr()

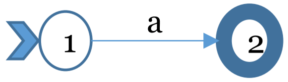
$\boxed{(\quad (\quad a}\quad)\quad *\quad)\ .\ (\quad (\quad b\quad)\quad *\quad)$

```
        consume LPAREN
        R1 = parse_expr()
                consume LPAREN
                R1  = parse_expr()
                        consume a
```

parse_expr()

$\boxed{(\quad (\quad a)}\quad)\quad *\quad)\ .\ (\quad (\quad b\quad)\quad *\quad)$

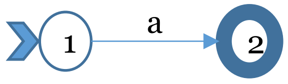
```
consume LPAREN
R1 = parse_expr()
          consume LPAREN
          R1  = parse_expr()
                    consume a
                    construct REG for a
```



parse_expr()

$\boxed{(\quad (\quad a}\quad)\quad *\quad)\;.\;(\quad (\quad b\quad)\quad *\quad)$

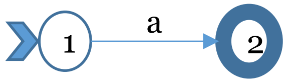
```
consume LPAREN
R1 = parse_expr()
          consume LPAREN
          R1  = parse_expr()
                    consume a
                    construct REG for a
                    return REG for a
```



parse_expr()

$\boxed{(\quad (\quad \underbrace{a}_{}}\quad)\quad *\quad)\ .\ (\quad (\quad b\quad)\quad *\quad)$

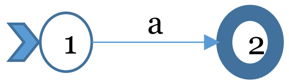
```
consume LPAREN
R1 = parse_expr()
          consume LPAREN
          R1  = parse_expr()   =
                    consume a
                    construct REG for a
                    return REG for a
```



parse_expr()

((a) *) . ((b) *)

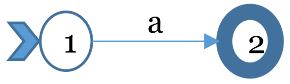
```
consume LPAREN
R1 = parse_expr()
        consume LPAREN
        R1 = parse_expr() =
                consume a
                construct REG for a
                return REG for a
        consume RPAREN
```



parse_expr()

$\boxed{(\quad(\quad a\quad)\quad *}\quad)\ .\ (\quad (\quad b\quad)\quad *\quad)$

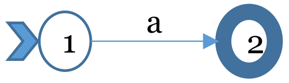
```
consume LPAREN
R1 = parse_expr()
          consume LPAREN
          R1  = parse_expr()   =
                    consume a
                    construct REG for a
                    return REG for a
          consume RPAREN
          consume STAR
```

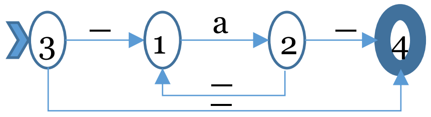


parse_expr()

$\boxed{(\quad (\quad a \quad) \quad *}$) . ((b) *)

```
consume LPAREN
R1 = parse_expr()
        consume LPAREN
        R1  = parse_expr()   =
                consume a
                construct REG for a
                return REG for a
        consume RPAREN
        consume STAR
        construct REG from (a)*
```

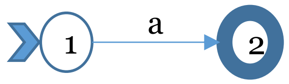


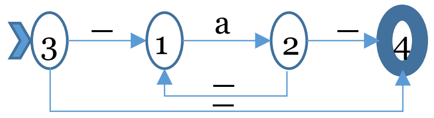


parse_expr()

$\boxed{(\quad(\quad a\quad)\quad *}\quad)\ .\ (\quad(\quad b\quad)\quad *\quad)$

```
consume LPAREN
R1 = parse_expr()
        consume LPAREN
        R1  = parse_expr()   =
                consume a
                construct REG for a
                return REG for a
        consume RPAREN
        consume STAR
        construct REG from (a)*
        return REG for (a)*
```

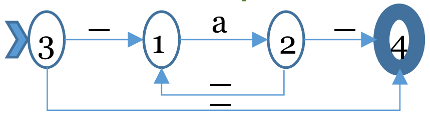




parse_expr()

$\boxed{(\quad \underbrace{(\quad a \quad) \quad *}_{}}\quad) \;.\; (\quad (\quad b \quad) \quad * \quad)$

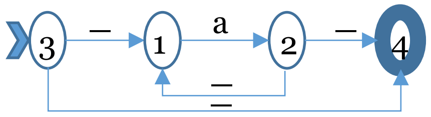
```
consume LPAREN
R1 = parse_expr()                =
        consume LPAREN
        R1  = parse_expr() =
                consume a
                construct REG for a
                return REG for a
        consume RPAREN
        consume STAR
        construct REG from (a)*
        return REG for (a)*
```



parse_expr()

((a) *) . ((b) *)

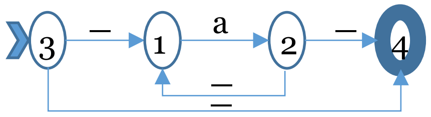
```
consume LPAREN
R1 = parse_expr()          =
        consume LPAREN
        R1  = parse_expr() =
                consume a
                construct REG for a
                return REG for a
        consume RPAREN
        consume STAR
        construct REG from (a)*
        return REG for (a)*
consume RPAREN
```



parse_expr()

((a) *) . ((b) *)

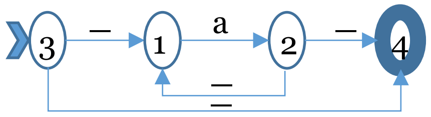
```
consume LPAREN
R1 = parse_expr()          =
        consume LPAREN
        R1  = parse_expr() =
                consume a
                construct REG for a
                return REG for a
        consume RPAREN
        consume STAR
        construct REG from (a)*
        return REG for (a)*
consume RPAREN
consume DOT
```



parse_expr()

((a) *) . ((b) *)

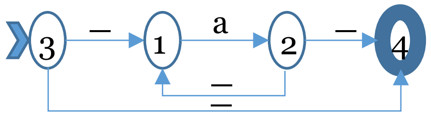
```
consume LPAREN
R1 = parse_expr()          =
          consume LPAREN
          R1  = parse_expr() =
                    consume a
                    construct REG for a
                    return REG for a
          consume RPAREN
          consume STAR
          construct REG from (a)*
          return REG for (a)*
consume RPAREN
consume DOT
consume LPAREN
```



parse_expr()

((a) *) . ((b) *)

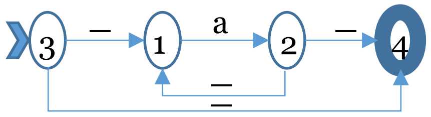
```
consume LPAREN
R1 = parse_expr()          =
          consume LPAREN
          R1  = parse_expr() =
                    consume a
                    construct REG for a
                    return REG for a
          consume RPAREN
          consume STAR
          construct REG from (a)*
          return REG for (a)*
consume RPAREN
consume DOT
consume LPAREN
R2 = parse_expr()
```



parse_expr()

((a) *) . ((b) *)

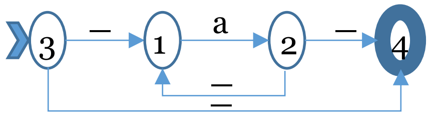
```
consume LPAREN
R1 = parse_expr()          =
          consume LPAREN
          R1  = parse_expr() =
                    consume a
                    construct REG for a
                    return REG for a
          consume RPAREN
          consume STAR
          construct REG from (a)*
          return REG for (a)*
consume RPAREN
consume DOT
consume LPAREN
R2 = parse_expr()
          consume LPAREN
```



parse_expr()

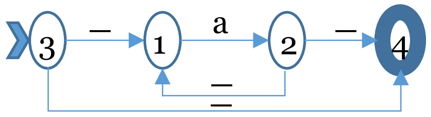
((a) *) . ((b) *)

```
consume LPAREN
R1 = parse_expr()          =
        consume LPAREN
        R1  = parse_expr() =
                consume a
                construct REG for a
                return REG for a
        consume RPAREN
        consume STAR
        construct REG from (a)*
        return REG for (a)*
consume RPAREN
consume DOT
consume LPAREN
R2 = parse_expr()
        consume LPAREN
        R1 = parse_expr()
```



parse_expr()

((a) *) . ((b) *)

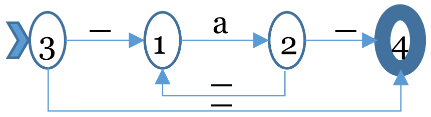


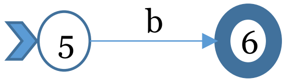
```
consume LPAREN
R1 = parse_expr()            =
        consume LPAREN
        R1  = parse_expr() =
                consume a
                construct REG for a
                return REG for a
        consume RPAREN
        consume STAR
        construct REG from (a)*
        return REG for (a)*
consume RPAREN
consume DOT
consume LPAREN
R2 = parse_expr()
        consume LPAREN
        R1 = parse_expr()
                consume b
```

parse_expr()

((a) *) . ((b) *)

```
consume LPAREN
R1 = parse_expr()            =
        consume LPAREN
        R1  = parse_expr() =
                consume a
                construct REG for a
                return REG for a
        consume RPAREN
        consume STAR
        construct REG from (a)*
        return REG for (a)*
consume RPAREN
consume DOT
consume LPAREN
R2 = parse_expr()
        consume LPAREN
        R1 = parse_expr()
                consume b
                construct REG for b
```

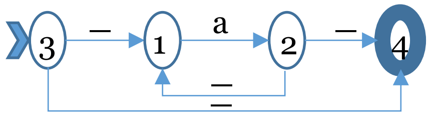


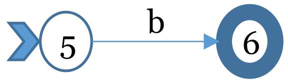


parse_expr()

((a) *) . ((b) *)

```
consume LPAREN
R1 = parse_expr()          =
        consume LPAREN
        R1  = parse_expr() =
                consume a
                construct REG for a
                return REG for a
        consume RPAREN
        consume STAR
        construct REG from (a)*
        return REG for (a)*
consume RPAREN
consume DOT
consume LPAREN
R2 = parse_expr()
        consume LPAREN
        R1 = parse_expr()
                consume b
                construct REG for b
                return REG for b
```

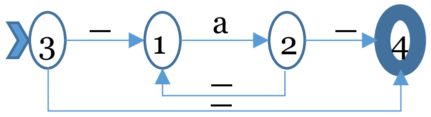


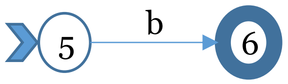


parse_expr()

((a) *) . ((b) *)

```
consume LPAREN
R1 = parse_expr()          =
        consume LPAREN
        R1  = parse_expr() =
                consume a
                construct REG for a
                return REG for a
        consume RPAREN
        consume STAR
        construct REG from (a)*
        return REG for (a)*
consume RPAREN
consume DOT
consume LPAREN
R2 = parse_expr()
        consume LPAREN
        R1 = parse_expr()        =
                consume b
                construct REG for b
                return REG for b
```

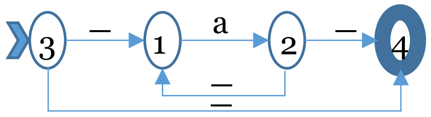


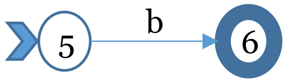


parse_expr()

((a) *) . ((b) *)

```
consume LPAREN
R1 = parse_expr()          =
        consume LPAREN
        R1  = parse_expr() =
                consume a
                construct REG for a
                return REG for a
        consume RPAREN
        consume STAR
        construct REG from (a)*
        return REG for (a)*
consume RPAREN
consume DOT
consume LPAREN
R2 = parse_expr()
        consume LPAREN
        R1 = parse_expr()      =
                consume b
                construct REG for b
                return REG for b
        consume RPAREN
```

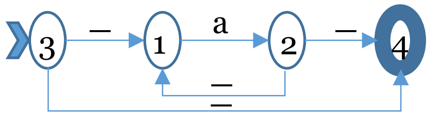


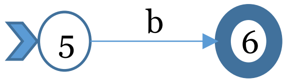


parse_expr()

((a) *) . ((b) *)

```
consume LPAREN
R1 = parse_expr()          =
        consume LPAREN
        R1  = parse_expr() =
                consume a
                construct REG for a
                return REG for a
        consume RPAREN
        consume STAR
        construct REG from (a)*
        return REG for (a)*
consume RPAREN
consume DOT
consume LPAREN
R2 = parse_expr()
        consume LPAREN
        R1 = parse_expr()      =
                consume b
                construct REG for b
                return REG for b
        consume RPAREN
        consume STAR
```

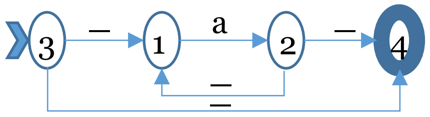


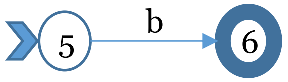


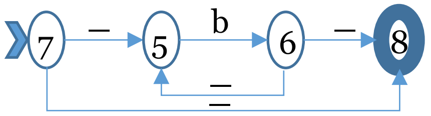
parse_expr()

((a) *) . ((b) *)

```
consume LPAREN
R1 = parse_expr()          =
        consume LPAREN
        R1 = parse_expr() =
                consume a
                construct REG for a
                return REG for a
        consume RPAREN
        consume STAR
        construct REG from (a)*
        return REG for (a)*
consume RPAREN
consume DOT
consume LPAREN
R2 = parse_expr()
        consume LPAREN
        R1 = parse_expr()       =
                consume b
                construct REG for b
                return REG for b
        consume RPAREN
        consume STAR
        construct REG for (b)*
```



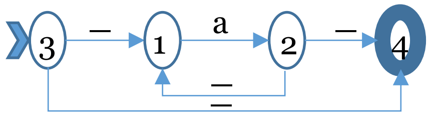




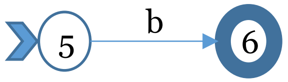
parse_expr()

((a) *) . ((b) *))

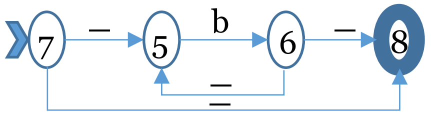
```
consume LPAREN
R1 = parse_expr()        =
        consume LPAREN
        R1  = parse_expr() =
                consume a
                construct REG for a
                return REG for a
        consume RPAREN
        consume STAR
        construct REG from (a)*
        return REG for (a)*
```



```
consume RPAREN
consume DOT
consume LPAREN
R2 = parse_expr()
        consume LPAREN
        R1 = parse_expr()        =
                consume b
                construct REG for b
                return REG for b
```



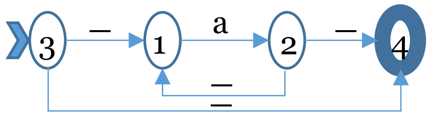
```
        consume RPAREN
        consume STAR
        construct REG for (b)*
        return REG for (b)*
```

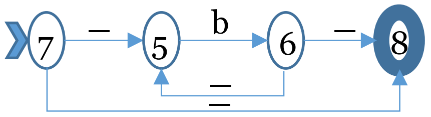


parse_expr()

((a) *) . ((b) *)

```
consume LPAREN
R1 = parse_expr()          =
        consume LPAREN
        R1  = parse_expr() =
                consume a
                construct REG for a
                return REG for a
        consume RPAREN
        consume STAR
        construct REG from (a)*
        return REG for (a)*
consume RPAREN
consume DOT
consume LPAREN
R2 = parse_expr()          =
        consume LPAREN
        R1 = parse_expr()      =
                consume b
                construct REG for b
                return REG for b
        consume RPAREN
        consume STAR
        construct REG for (b)*
        return REG for (b)*
```

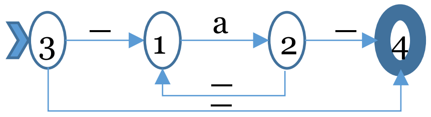


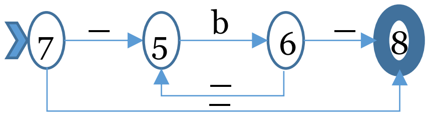


parse_expr()

((a) *) . ((b) *)

```
consume LPAREN
R1 = parse_expr()          =
        consume LPAREN
        R1  = parse_expr() =
                consume a
                construct REG for a
                return REG for a
        consume RPAREN
        consume STAR
        construct REG from (a)*
        return REG for (a)*
consume RPAREN
consume DOT
consume LPAREN
R2 = parse_expr()          =
        consume LPAREN
        R1 = parse_expr()       =
                consume b
                construct REG for b
                return REG for b
        consume RPAREN
        consume STAR
        construct REG for (b)*
        return REG for (b)*
consume RPAREN
```

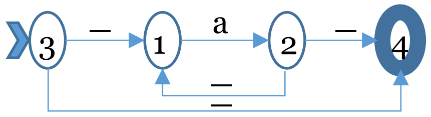


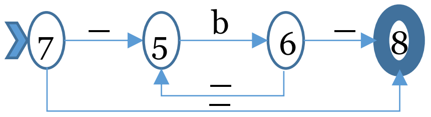


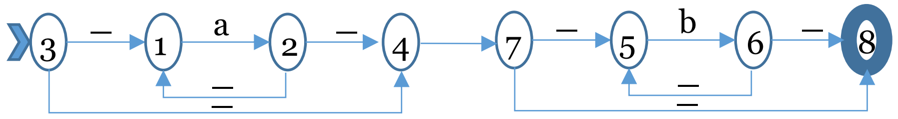
parse_expr()

((a) *) . ((b) *)

```
consume LPAREN
R1 = parse_expr()            =
          consume LPAREN
          R1  = parse_expr() =
                    consume a
                    construct REG for a
                    return REG for a
          consume RPAREN
          consume STAR
          construct REG from (a)*
          return REG for (a)*
consume RPAREN
consume DOT
consume LPAREN
R2 = parse_expr()         =
          consume LPAREN
          R1 = parse_expr()       =
                    consume b
                    construct REG for b
                    return REG for b
          consume RPAREN
          consume STAR
          construct REG for (b)*
          return REG for (b)*
consume RPAREN
construct REG for ((a*).((b)*)
```



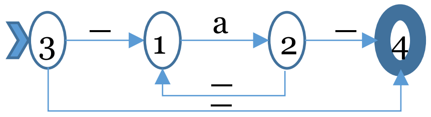


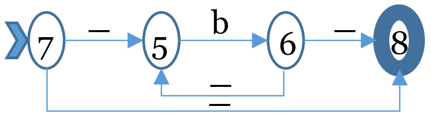


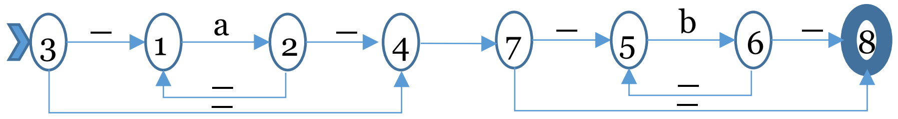
parse_expr()

((a) *) . ((b) *)

```
consume LPAREN
R1 = parse_expr()          =
        consume LPAREN
        R1  = parse_expr() =
                consume a
                construct REG for a
                return REG for a
        consume RPAREN
        consume STAR
        construct REG from (a)*
        return REG for (a)*
consume RPAREN
consume DOT
consume LPAREN
R2 = parse_expr()          =
        consume LPAREN
        R1 = parse_expr()      =
                consume b
                construct REG for b
                return REG for b
        consume RPAREN
        consume STAR
        construct REG for (b)*
        return REG for (b)*
consume RPAREN
construct REG for ((a*).((b)*)
return REG for ((a)*).((b)*)
```







How to match strings?

So far, we have seen how to construct REGs. Next, I will show how to determine if a string belongs to the language of a regular expression

1. I will start by defining paths a
2. Then I will define accepting path
3. Then, I will state the MAIN theorem for REGs which allows us to determine if a string is in the language of a regular expression by looking for an accepting path in the REG of the expression.
4. Then I will give examples of paths for a specific regular expression

How to match strings?

After I am done illustrating path, I will show an equivalent formulation in terms of reachable nodes

5. I will start by introducing reachable nodes state the equivalence between finding accepting paths and reachable nodes

6. Then I will give the condition for determining if a string in in the language of a regular expression in terms of reachable nodes

7. Then I will give examples of how reachable nodes are calculated (the pseudocode is given in the project description)

8. I will conclude by summarizing how longest matching prefix is found using reachable nodes

1. Definition of a path

- A ***path*** in a REG *r* is a sequence of nodes n_1 n_2 … n_k starting from the starting node of *r* such that consecutive nodes in the sequence are adjacent in the REG:

  - n_1 is the starting node of *r*
  - for each consecutive nodes n_i and n_{i+1}, there is an edge from n_i to n_{i+1} in *r*

- The ***string of a path*** is the string obtained by concatenating all labels on the edges of the path

2. Definition of accepting path

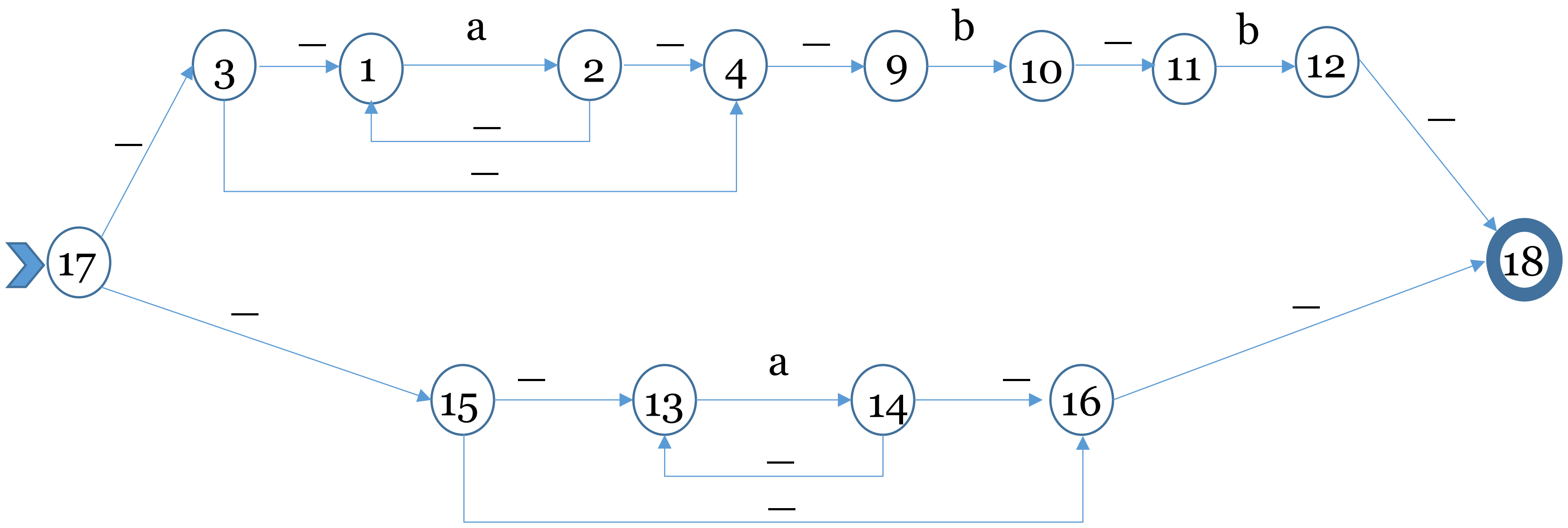
- An ***accepting path*** in a REG *r* is a sequence of nodes $n_1\ n_2\ \ldots\ n_k$ starting from the starting node of *r* and ending in the accepting node of *r* such that consecutive nodes in the sequence are adjacent in the REG:
  - n_1 is the starting node of *r*
  - n_k is the accepting node of *r*
  - for each consecutive nodes n_i and n_{i+1}, there is an edge from n_i to n_{i+1} in *r*
- The ***string of an accepting path*** is the string obtained by concatenating all labels on the edges of the accepting path

3. MAIN Theorem for REGs

- Let *R* be a regular expression and *r* be its REG

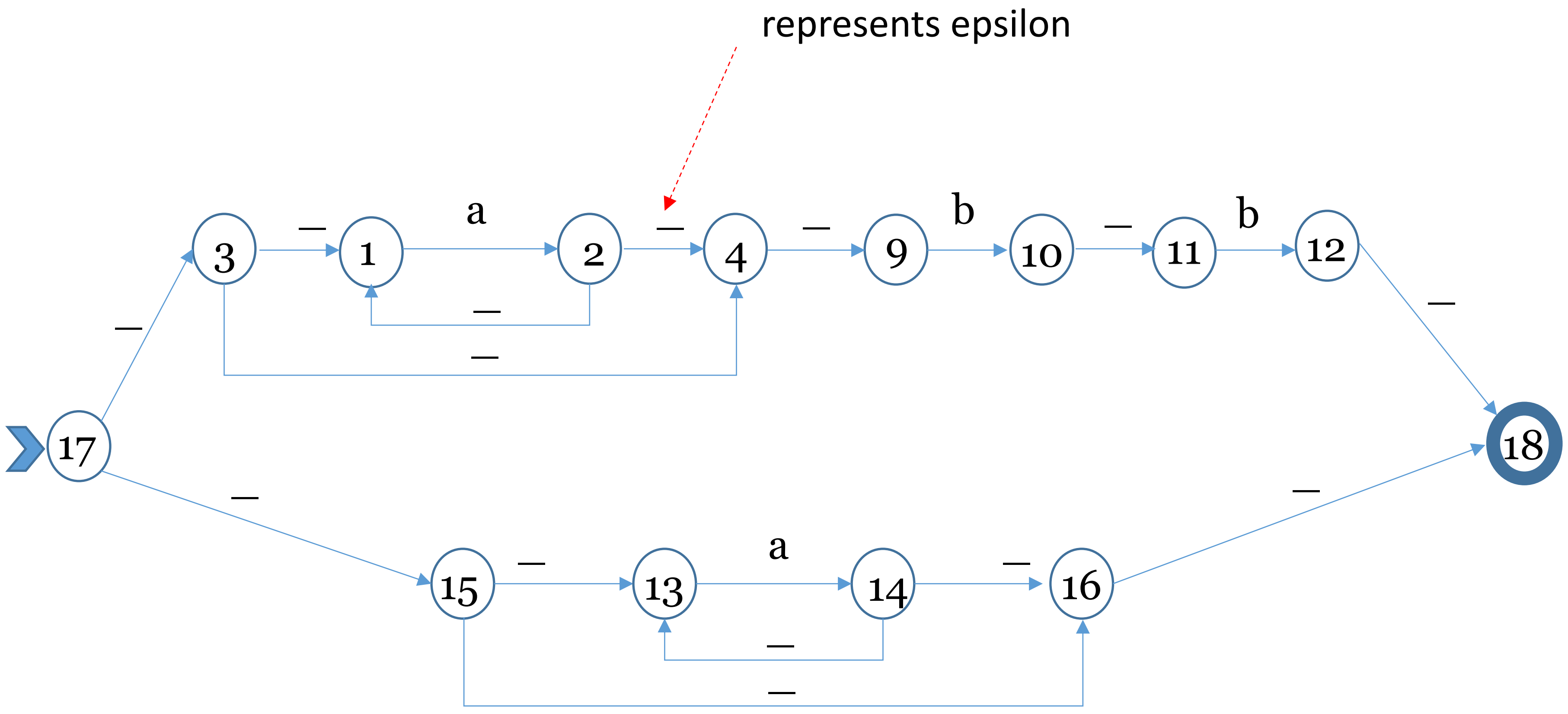
- **Theorem**: A string *w* is in L(*R*) if and only if there is an accepting path whose string is *w*

4. Example of path



REG for (((a)*). ((b).(b))) | ((a)*)

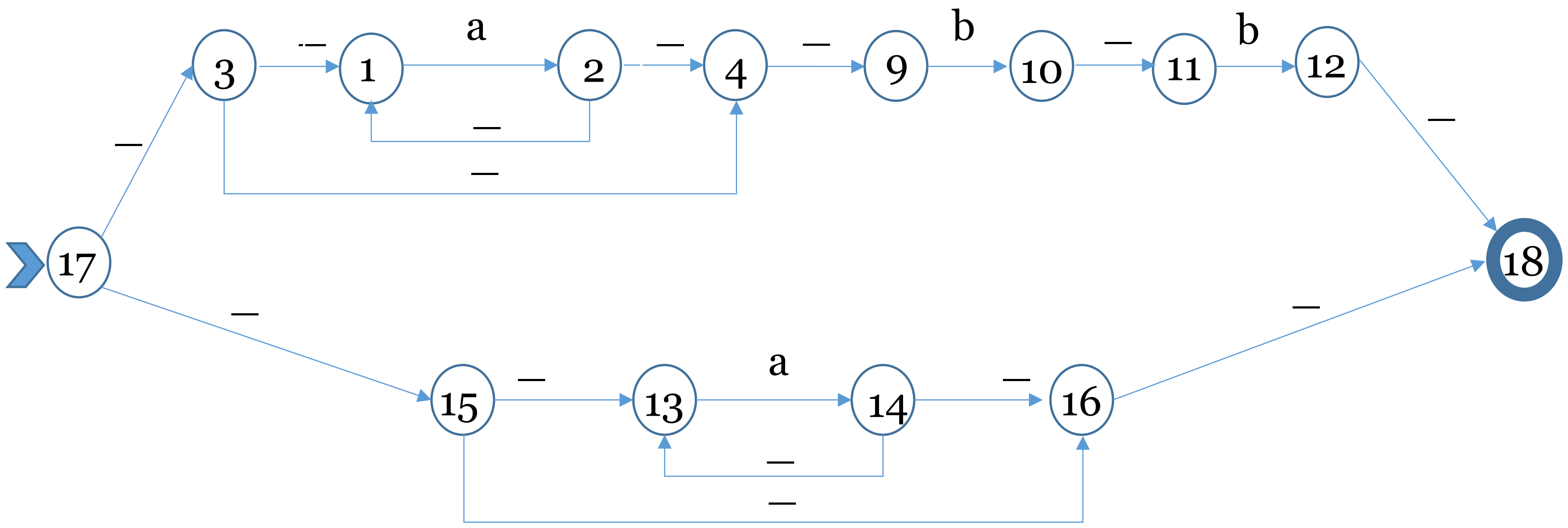
4. Example of path



REG for (((a)*). ((b).(b))) | ((a)*)

Examples of paths

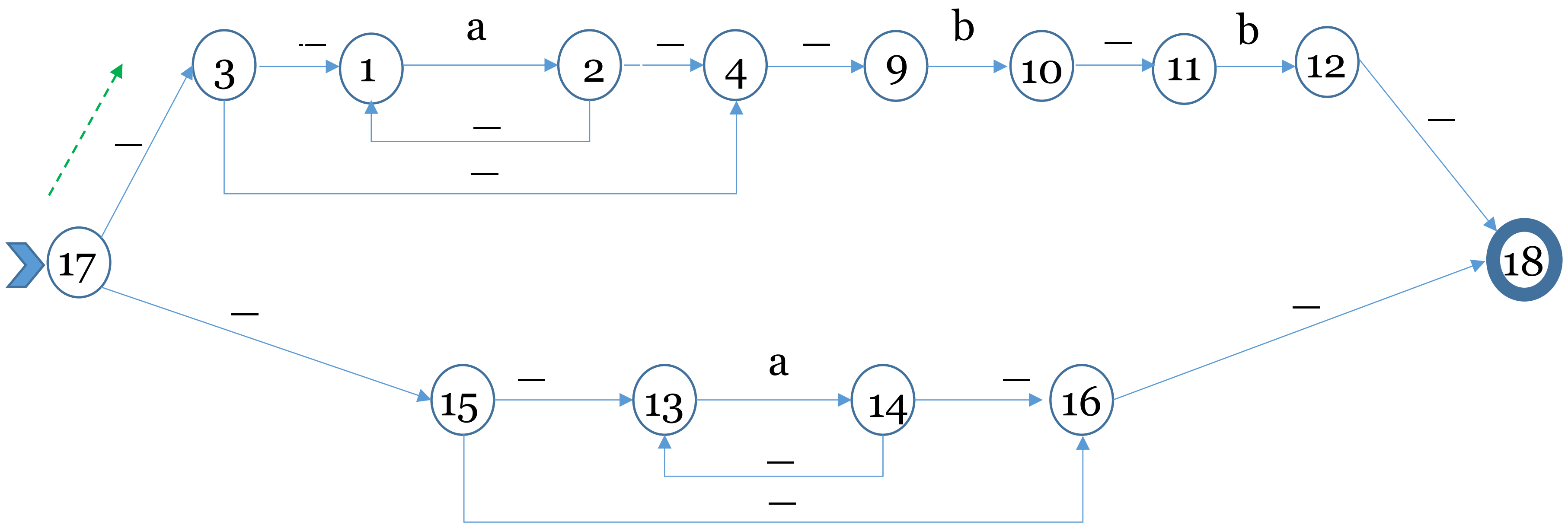
aabb =



REG for (((a)*). ((b).(b))) | ((a)*)

Examples of paths

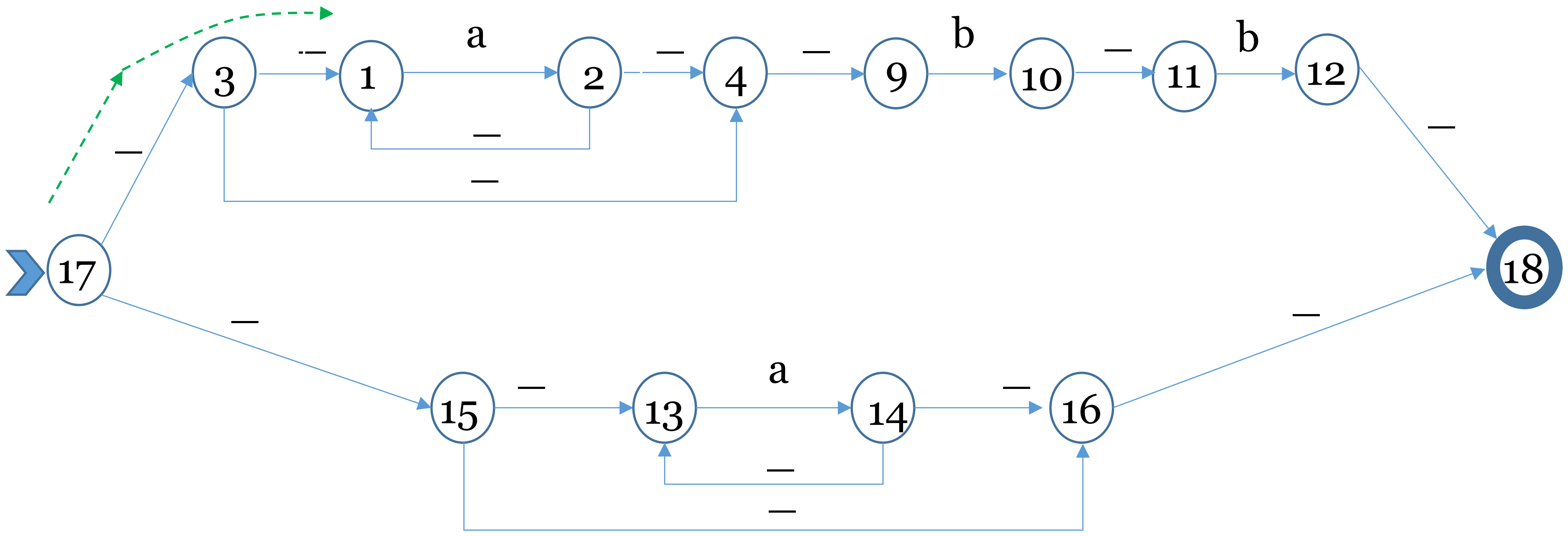
aabb = _



REG for (((a)*). ((b).(b))) | ((a)*)

Examples of paths

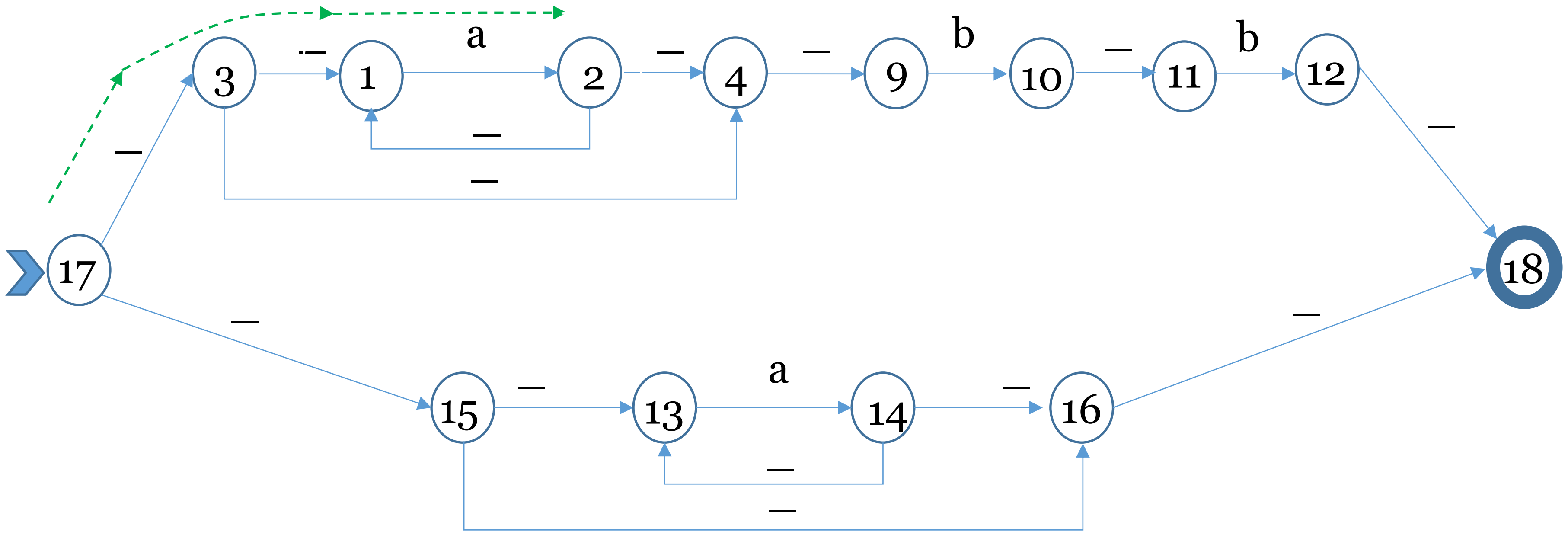
aabb = _ _



REG for (((a)*). ((b).(b))) | ((a)*)

Examples of paths

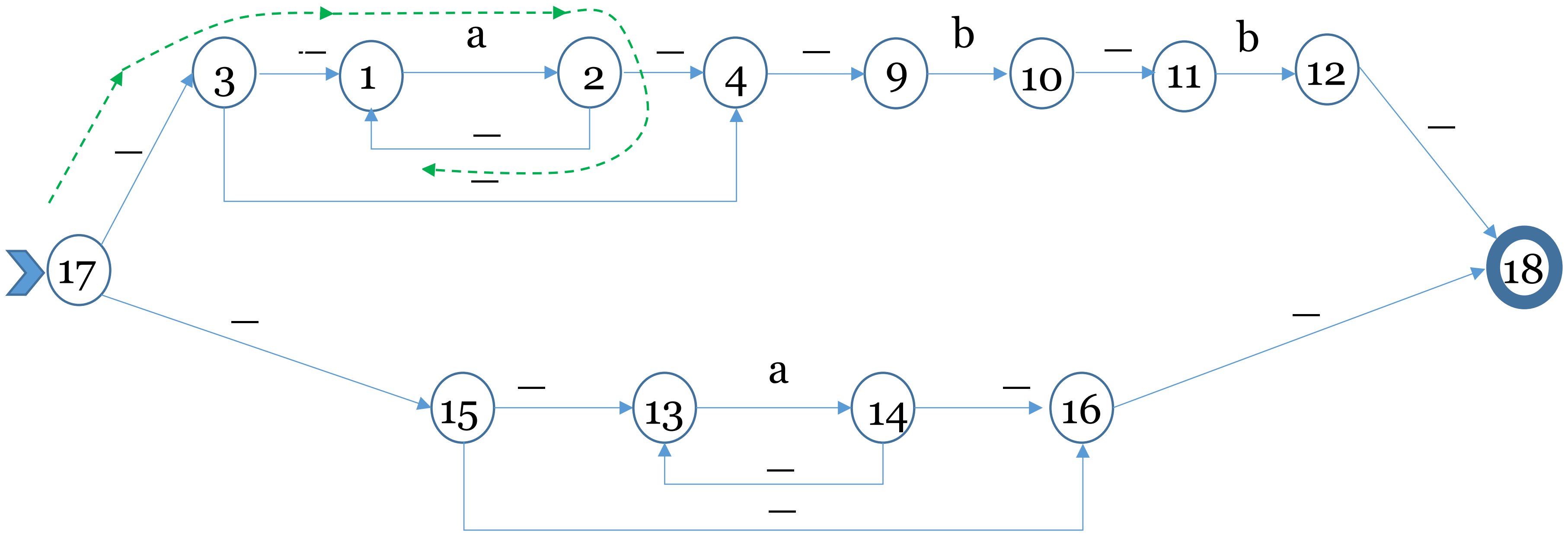
aabb = _ _ a



REG for (((a)*). ((b).(b))) | ((a)*)

Examples of paths

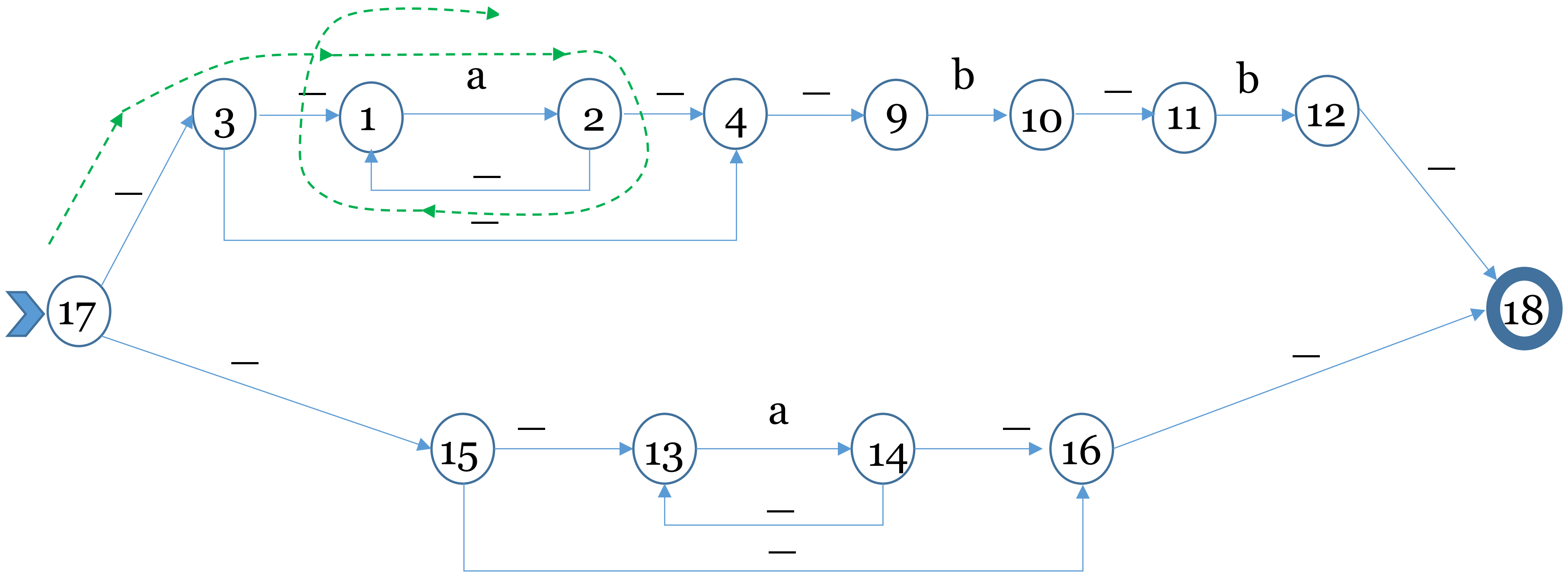
aabb = _ _ a _



REG for (((a)*). ((b).(b))) | ((a)*)

Examples of paths

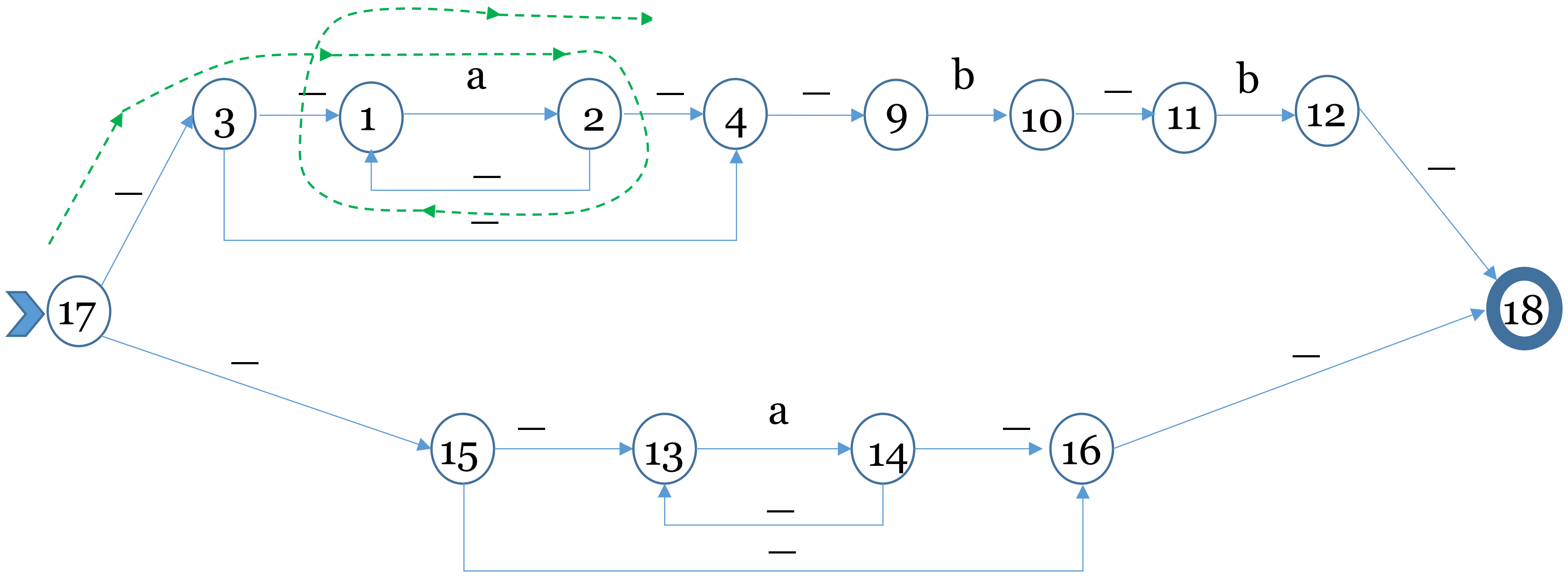
aabb = _ _ a _ a



REG for (((a)*). ((b).(b))) | ((a)*)

Examples of paths

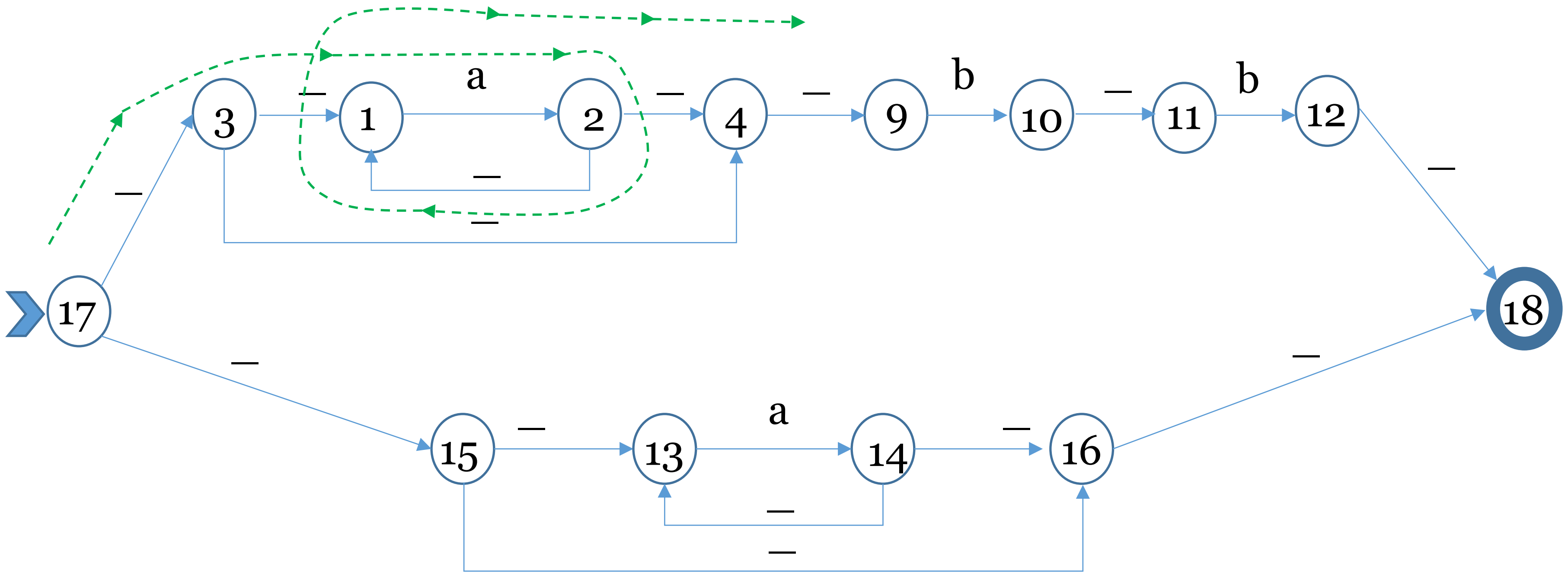
aabb = _ _ a _ a _



REG for (((a)*). ((b).(b))) | ((a)*)

Examples of paths

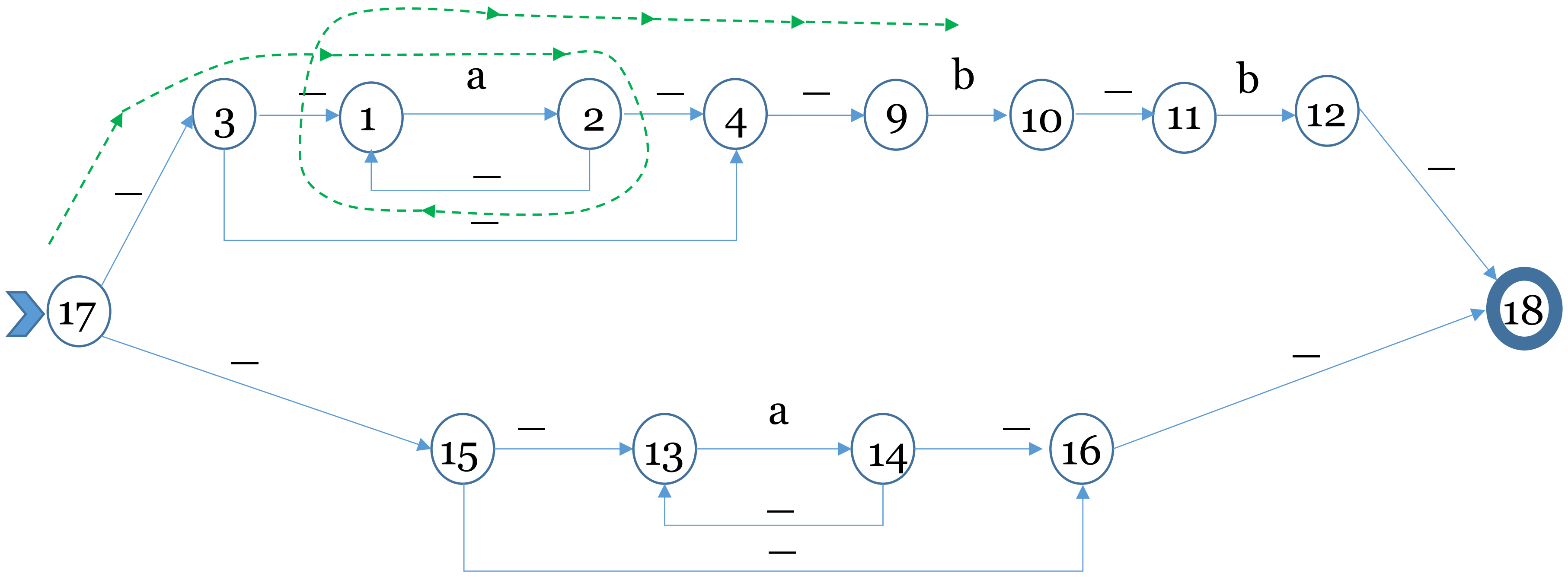
aabb = _ _ a _ a _ _ _



REG for (((a)*). ((b).(b))) | ((a)*)

Examples of paths

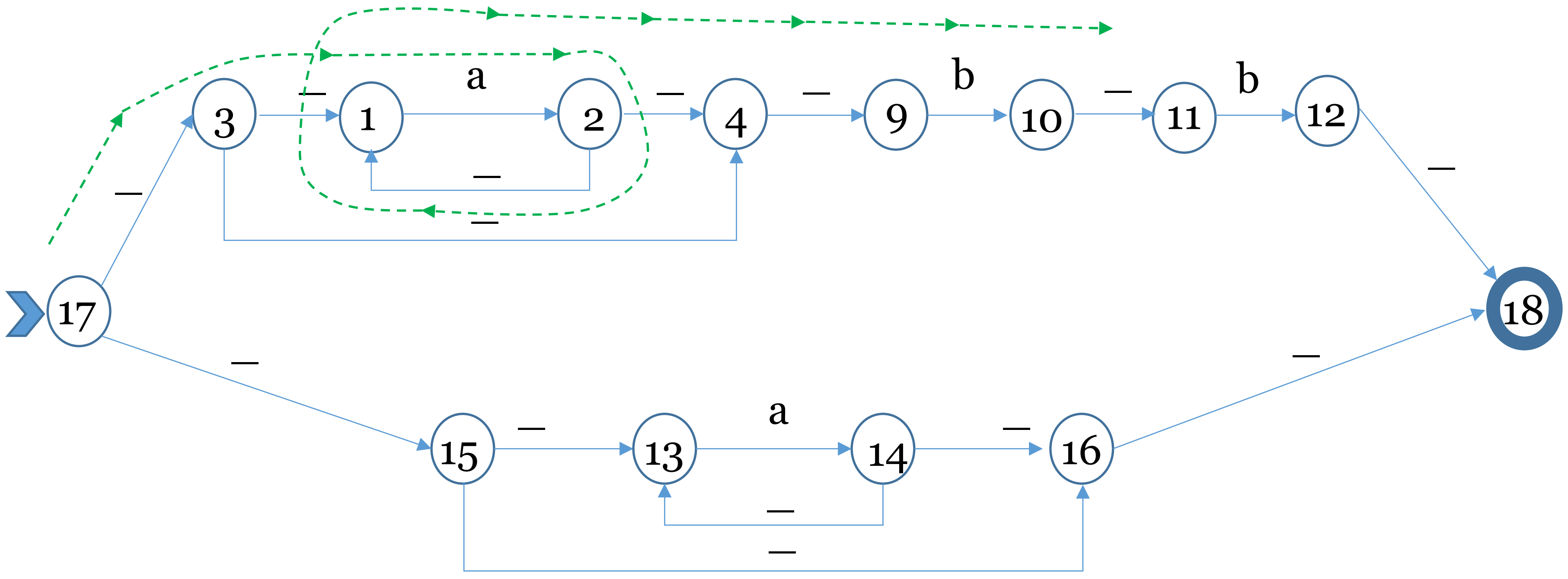
aabb = _ _ a _ a _ _ _ b



REG for (((a)*). ((b).(b))) | ((a)*)

Examples of paths

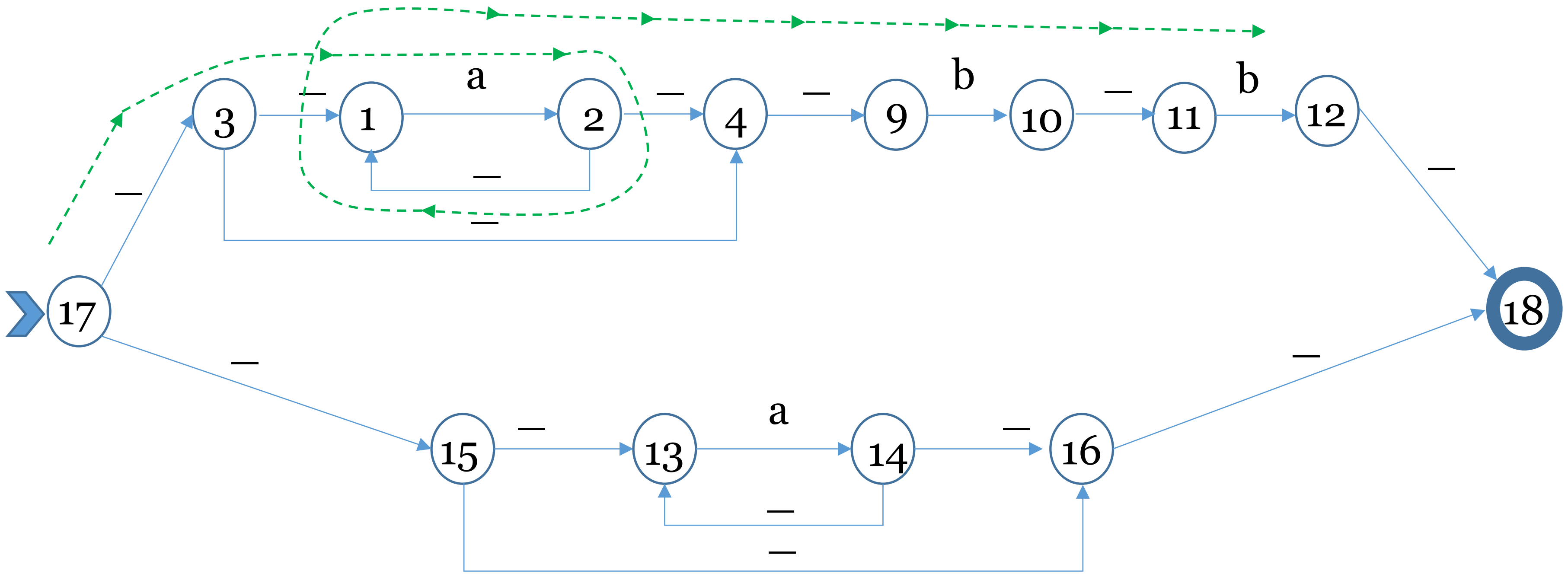
aabb = _ _ a _ a _ _ _ b _



REG for (((a)*). ((b).(b))) | ((a)*)

Examples of paths

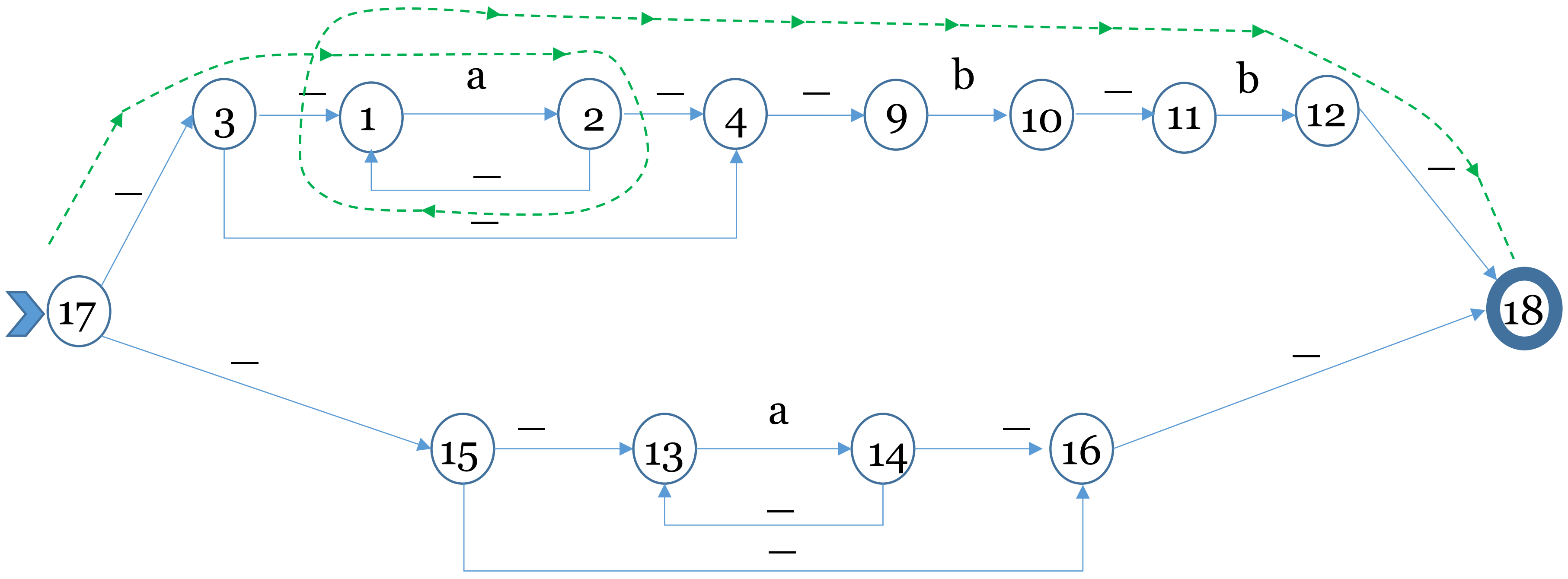
aabb = _ _ a _ a _ _ _ b _ b



REG for (((a)*). ((b).(b))) | ((a)*)

Examples of paths

aabb = _ _ a _ a _ _ b _ b _

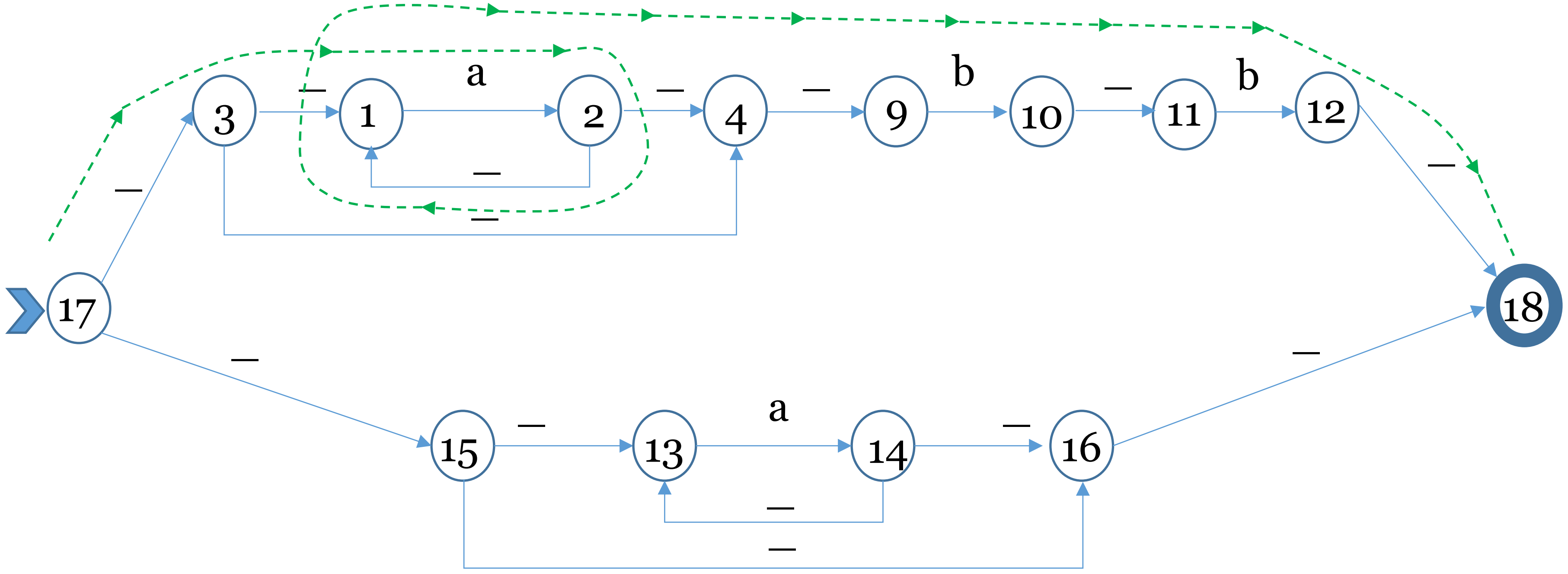


REG for (((a)*). ((b).(b))) | ((a)*)

Examples of paths

all of them are epsilon

aabb = _ _ a _ a _ _ b _ b _



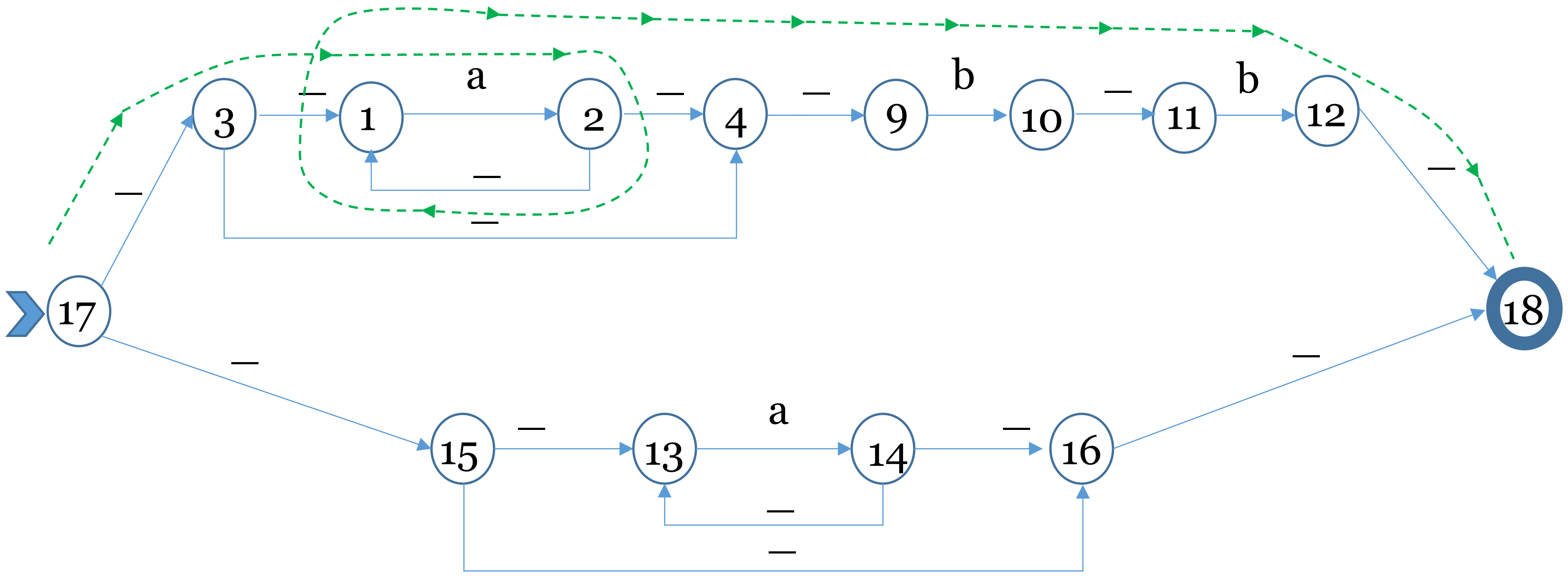
REG for (((a)*). ((b).(b))) | ((a)*)

Examples of paths

all of them are epsilon

aabb = _ _ a _ a _ _ b _ b _

aabb is accepted!

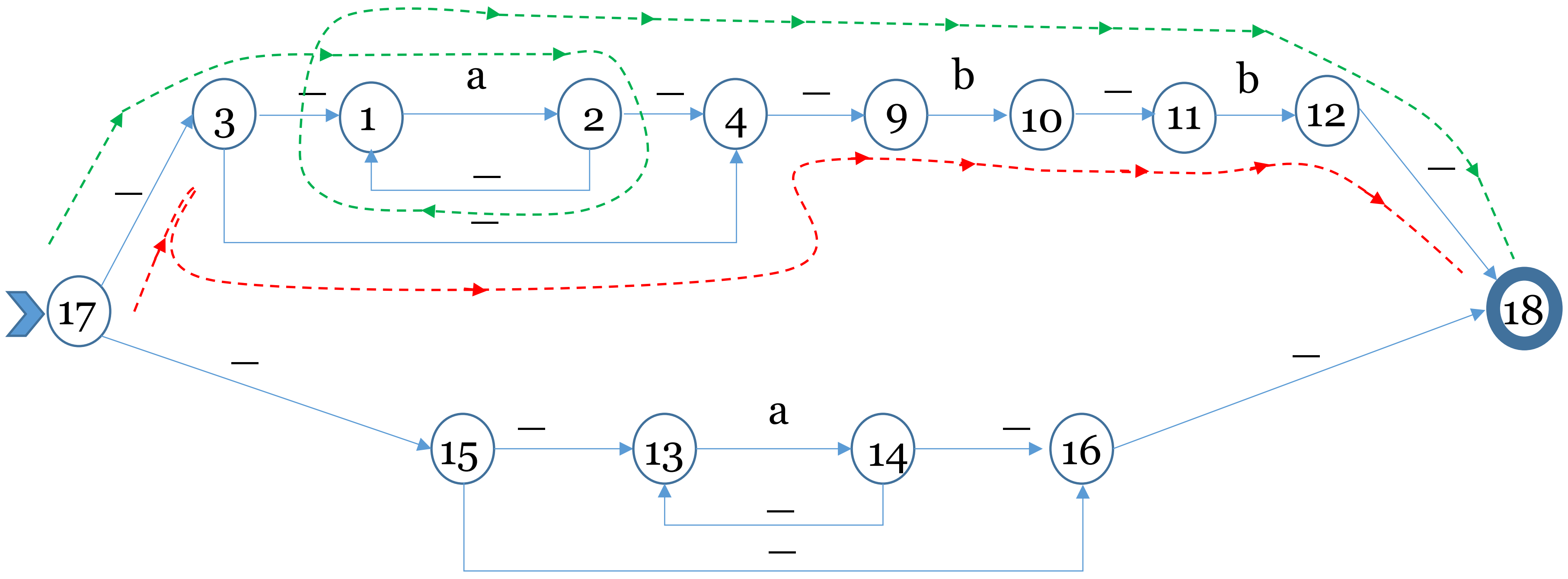


REG for (((a)*). ((b).(b))) | ((a)*)

Examples of paths

aabb = _ _ a _ a _ _ b _ b _

bb = _ _ _ b _ b _

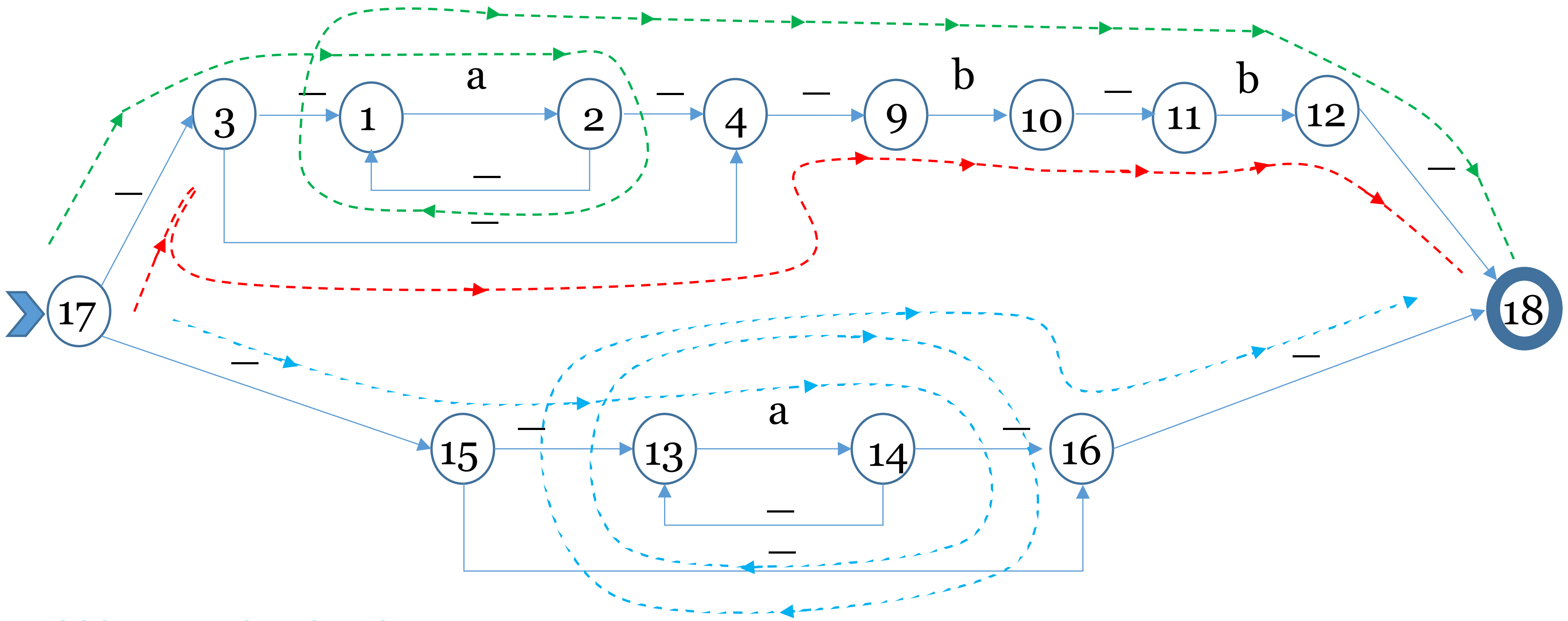


REG for (((a)*). ((b).(b))) | ((a)*)

Examples of paths

aabb = _ _ a _ a _ _ b _ b _

bb = _ _ _ b _ b _



aaa = _ _ a _ a _ a _ _

REG for (((a)*). ((b).(b))) | ((a)*)

5. Definition of reachable nodes

- For a path $n_1\ n_2\ \ldots\ n_k$ whose string is w, we say that n_k is **reachable by consuming** w

- If w is the string of an accepting path, then the accepting node is reachable by consuming w.

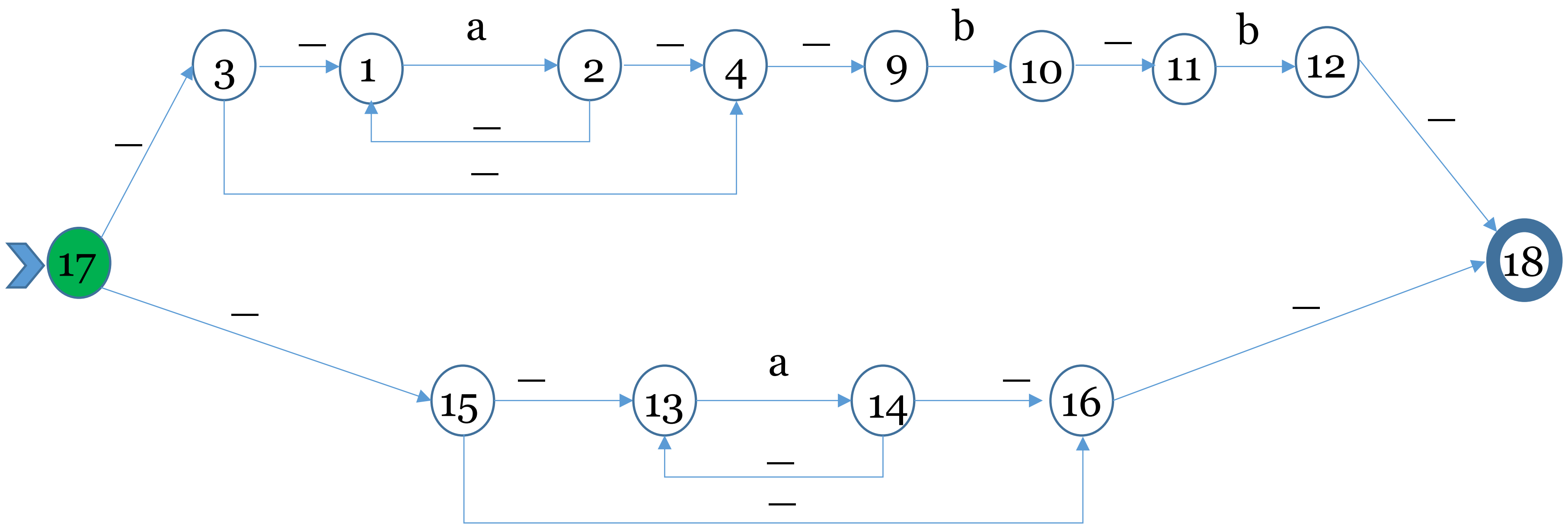
6. Restating the MAIN Theorem for REGs

- Let *R* be a regular expression and *r* be its REG

- **Theorem**: A string *w* is in L(*R*) if and only if the accepting node of *r* , the REG of *R* , is reachable by consuming *w*

7. Examples of reachable nodes

Input aba = _ a _ b _ a

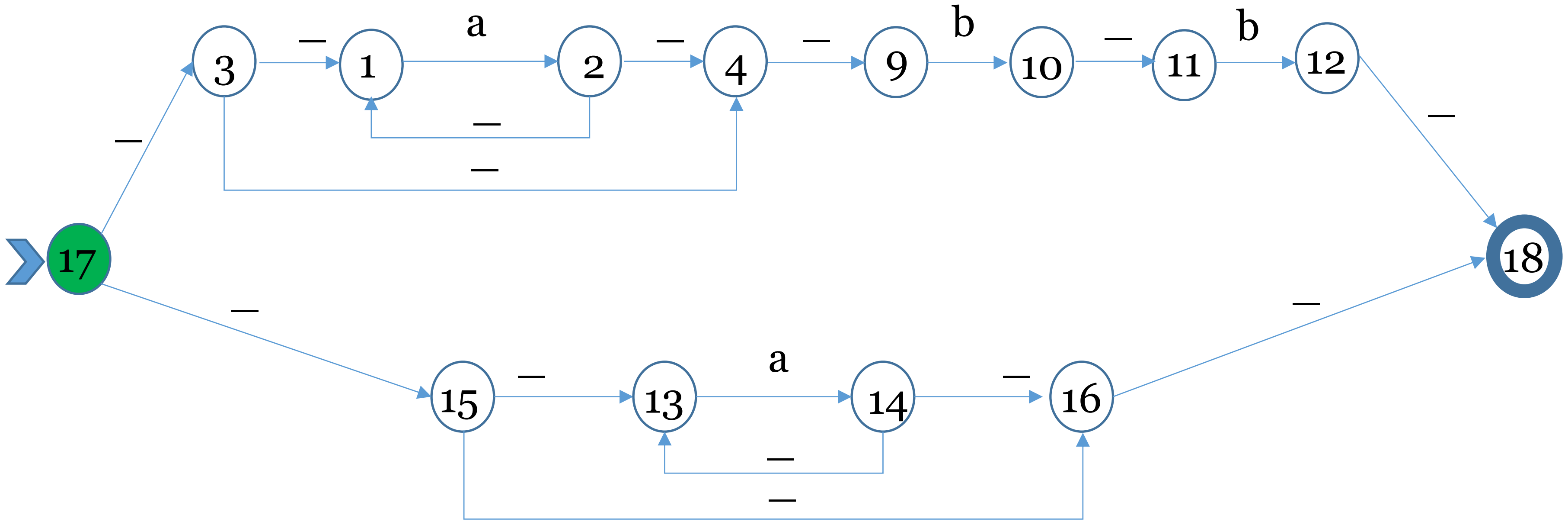


REG for (((a)*). ((b).(b))) | ((a)*)

Examples of reachable nodes

$\xrightarrow{\_}$ { 17,

Initially state 17 is reachable by consuming nothing.
It is the initial state

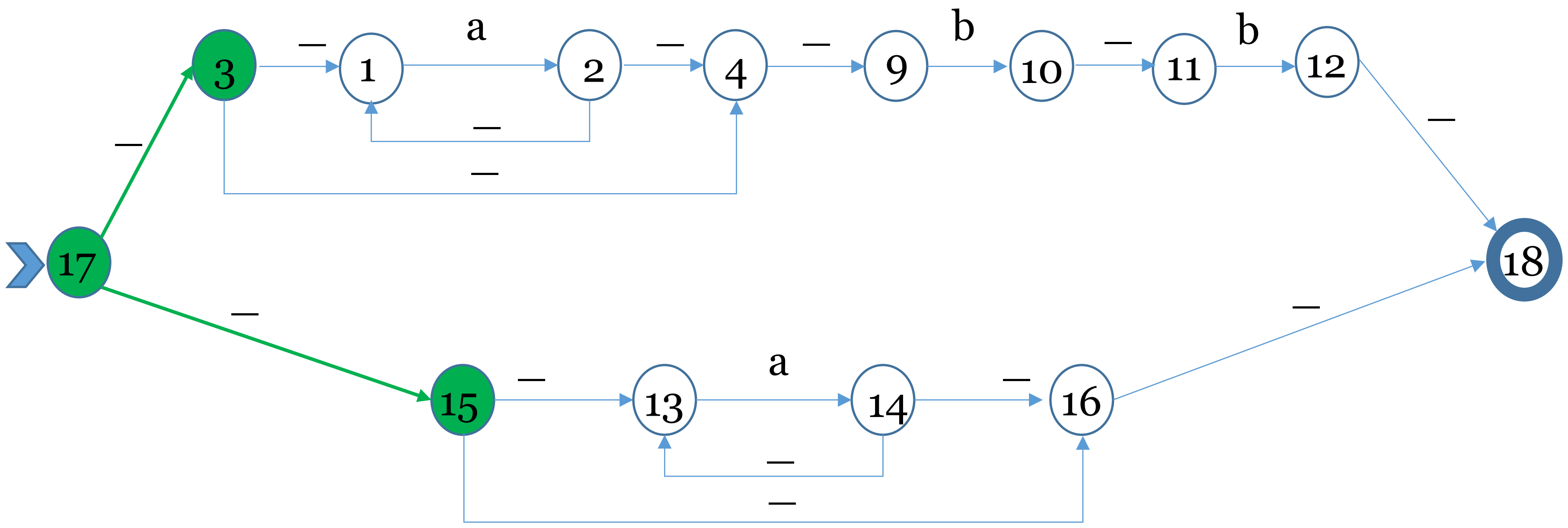


REG for (((a)*). ((b).(b))) | ((a)*)

Examples of reachable nodes

$\xrightarrow{-}$ { 17, 3 , 15

From state 17 is we can go to states 3 and 15 by consuming nothing

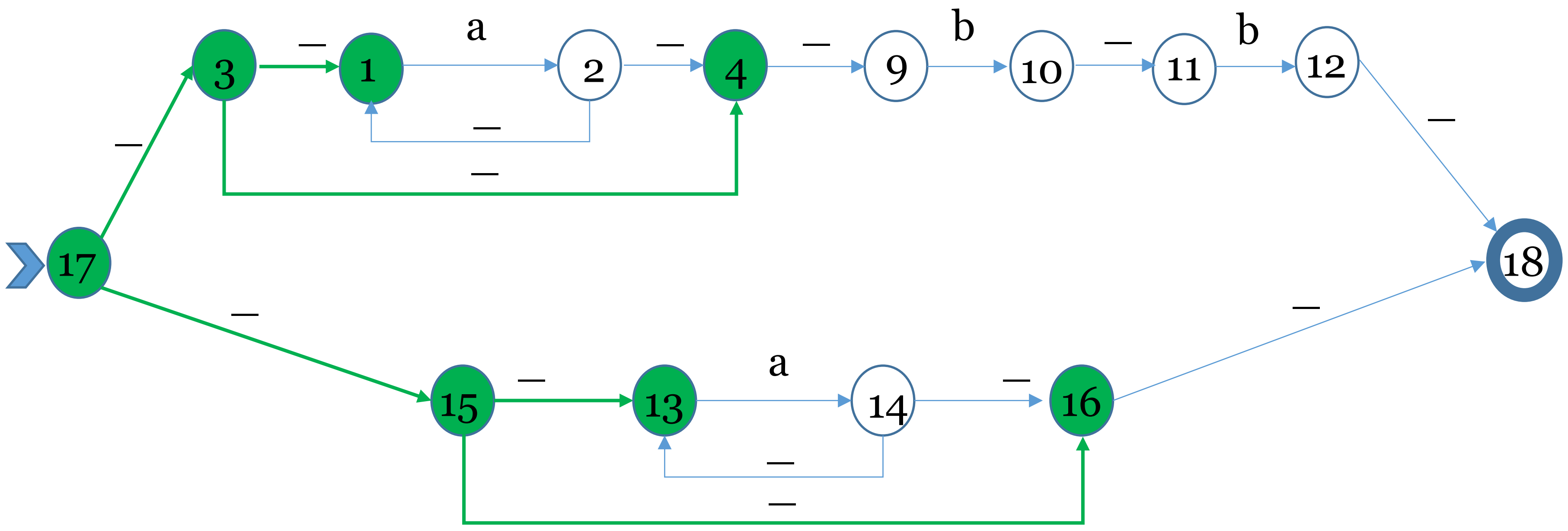


REG for (((a)*). ((b).(b))) | ((a)*)

Examples of reachable nodes

$\xrightarrow{\_}$ { 17, 3 , 15 , 1, 4, 13, 16

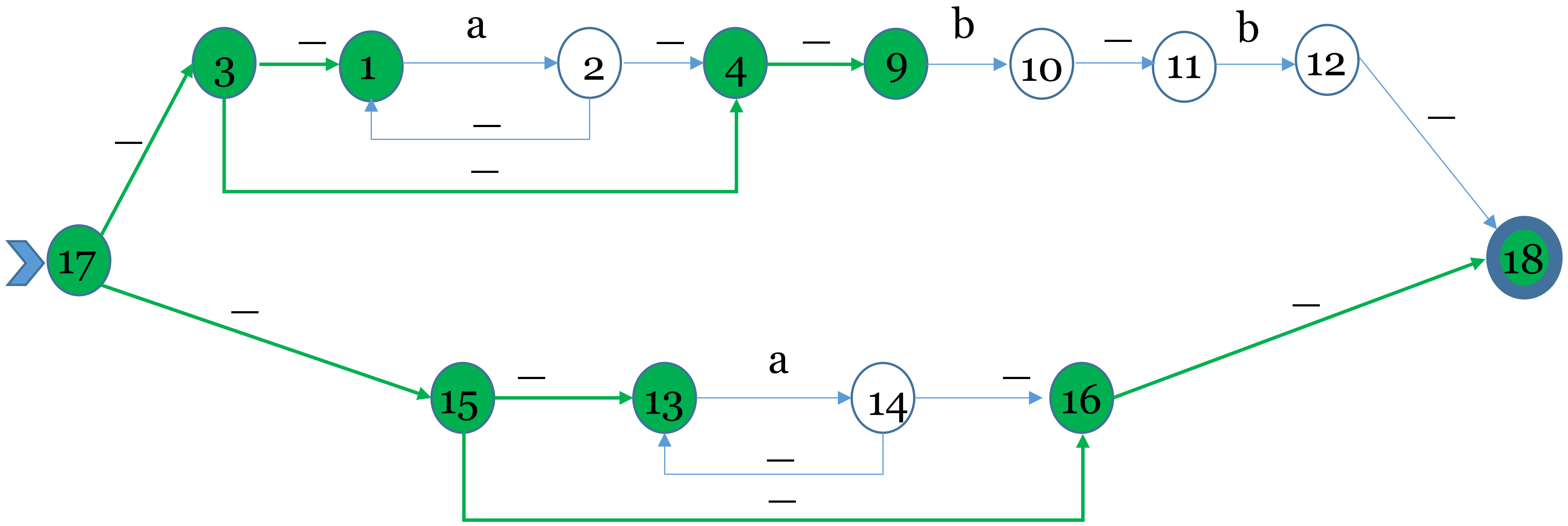
we can go from state 17 to states 3, 15 1, 4 , 13 and 16 by consuming nothing



REG for (((a)*). ((b).(b))) | ((a)*)

Examples of reachable nodes

$\xrightarrow{\_}$ { 17, 3 , 15 , 1, 4, 13, 16 , 9, **18** }

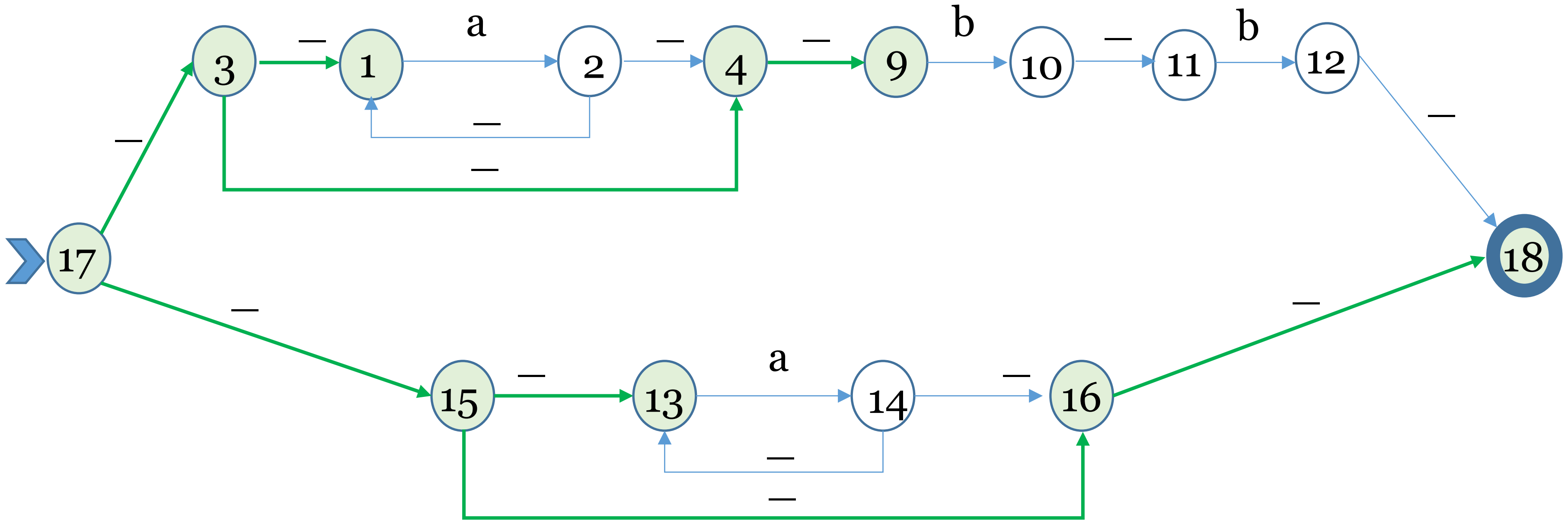


REG for (((a)*). ((b).(b))) | ((a)*)

Examples of reachable nodes

$\xrightarrow{\_}$ { 17, 3 , 15 , 1, 4, 13, 16 , 9, **18** }

The set represented by this regular expression contains epsilon because accept state **18** is reachable from state 17 without consuming any input

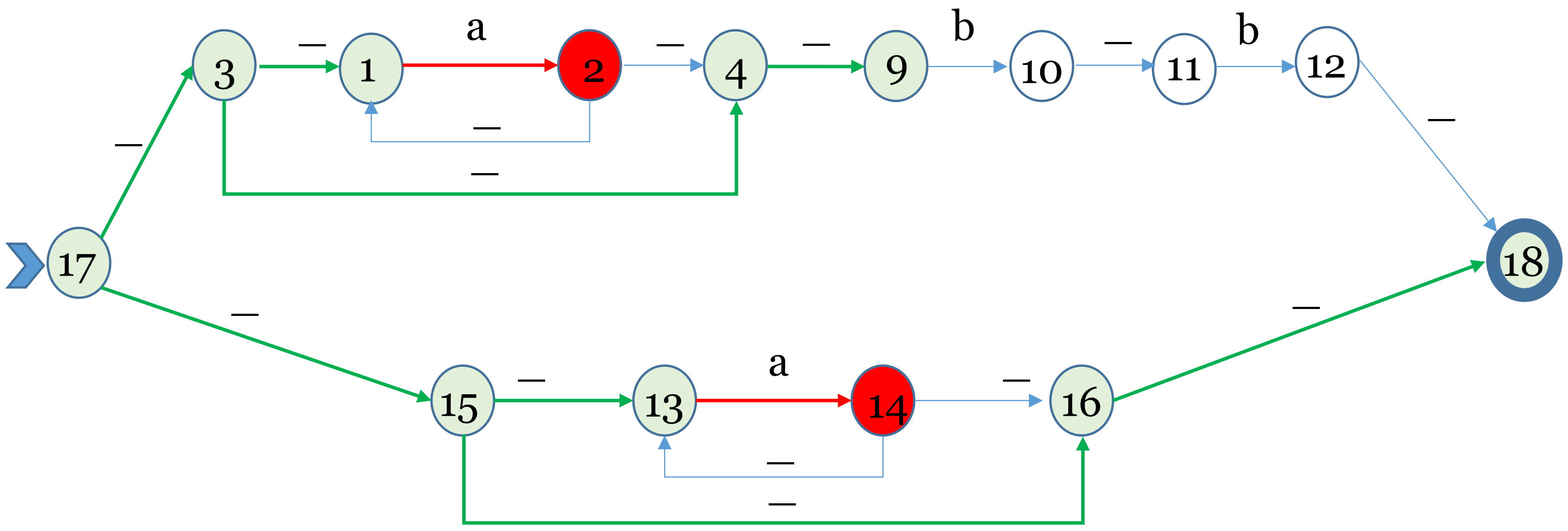


REG for (((a)*). ((b).(b))) | ((a)*)

Examples of reachable nodes

$\xrightarrow{\_} \{ 17, 3, 15, 1, 4, 13, 16, 9, \mathbf{18} \} \xrightarrow{a} \{ 2 , 14 \}$

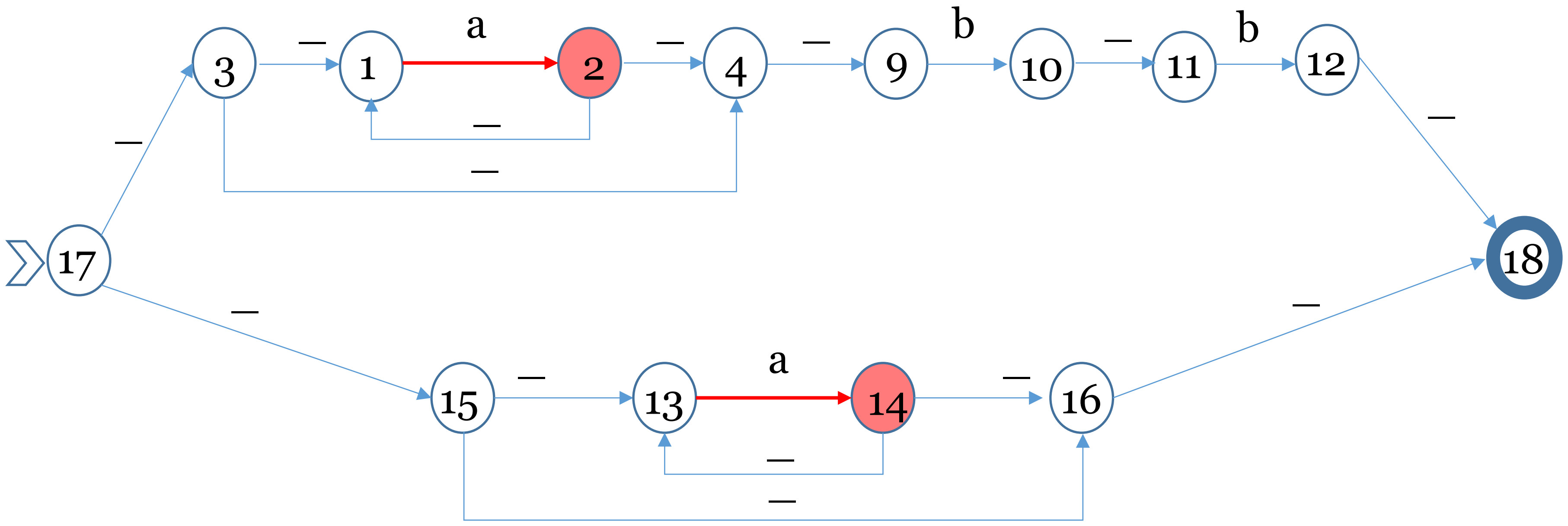
from the set of nodes that we obtained, we can go to the set { 2 , 14 } by consuming a



REG for (((a)*). ((b).(b))) | ((a)*)

Examples of reachable nodes

$\xrightarrow{\_} \{ 17, 3, 15, 1, 4, 13, 16, 9, \mathbf{18} \} \xrightarrow{a} \{ 2 , 14 \}$

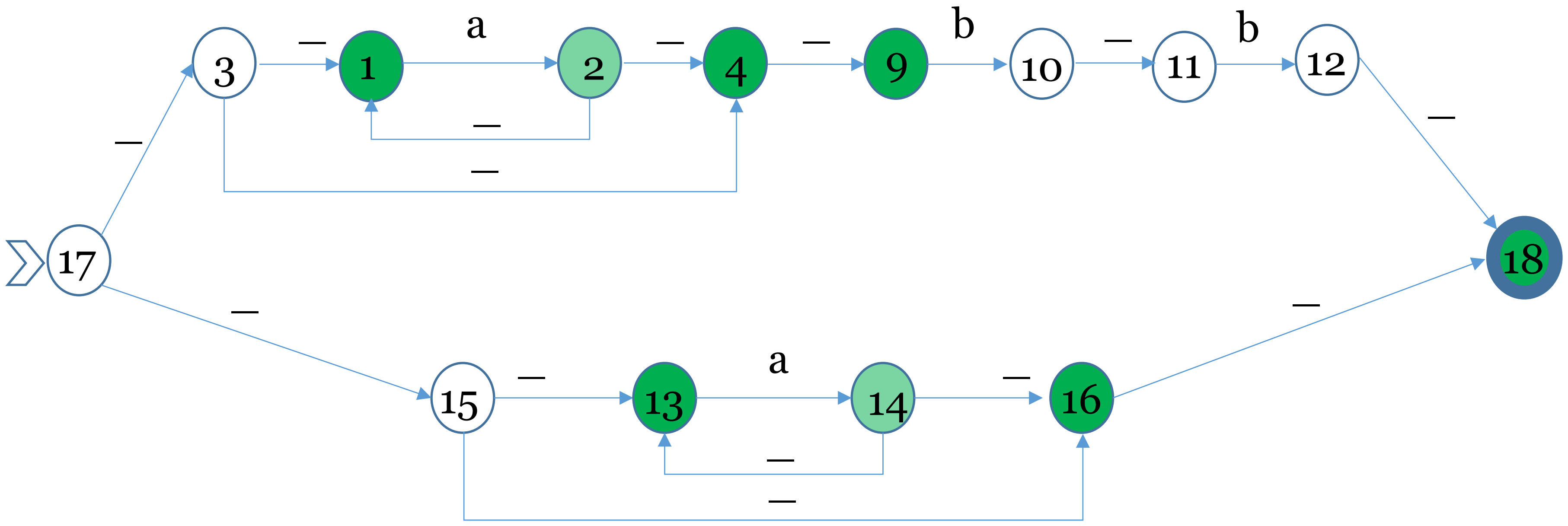


REG for (((a)*). ((b).(b))) | ((a)*)

Examples of reachable nodes

$\xrightarrow{\_}$ { 17, 3, 15, 1, 4, 13, 16, 9, **18** } $\xrightarrow{a}$ { 2 , 14 } $\xrightarrow{\_}$ { 1 , 2 , 4, 9, 13, 14, 16, **18**}

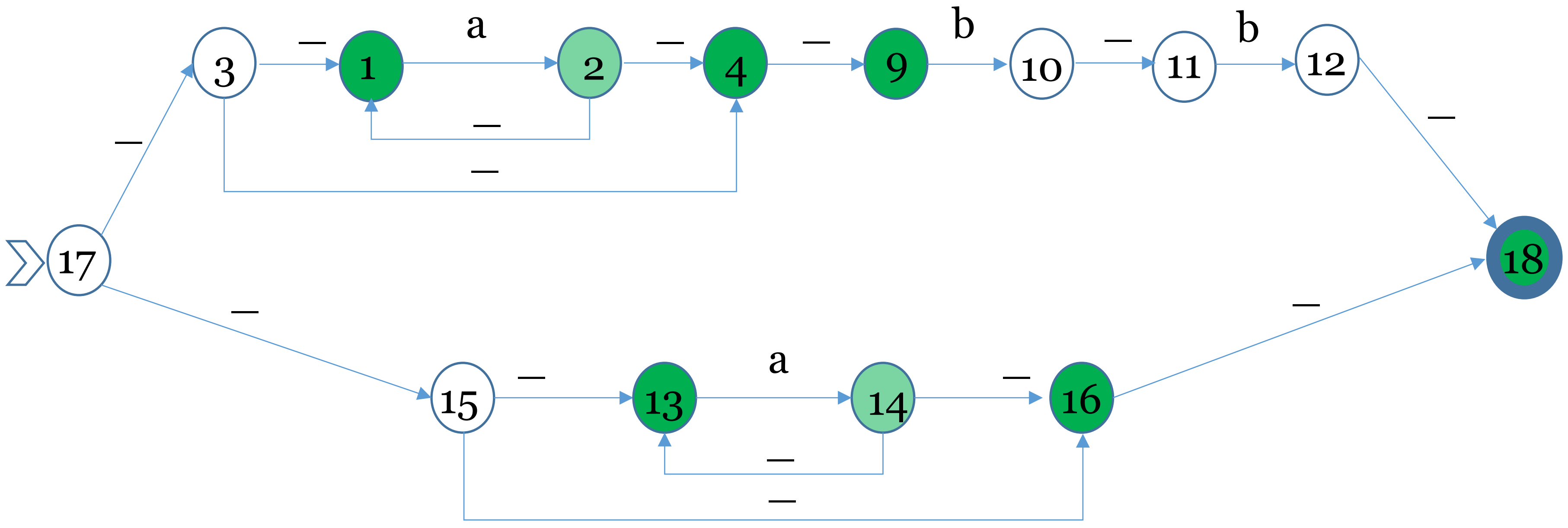
Then we can go to the set of nodes { 1 , 2 , 4, 9, 13, 14, 16, **18**} by consuming nothing



REG for (((a)*). ((b).(b))) | ((a)*)

$\xrightarrow{\_}$ { 17, 3, 15, 1, 4, 13, 16, 9, **18** } $\xrightarrow{a}$ { 2 , 14 } $\xrightarrow{\_}$ { 1 , 2 , 4, 9, 13, 14, 16, **18**}

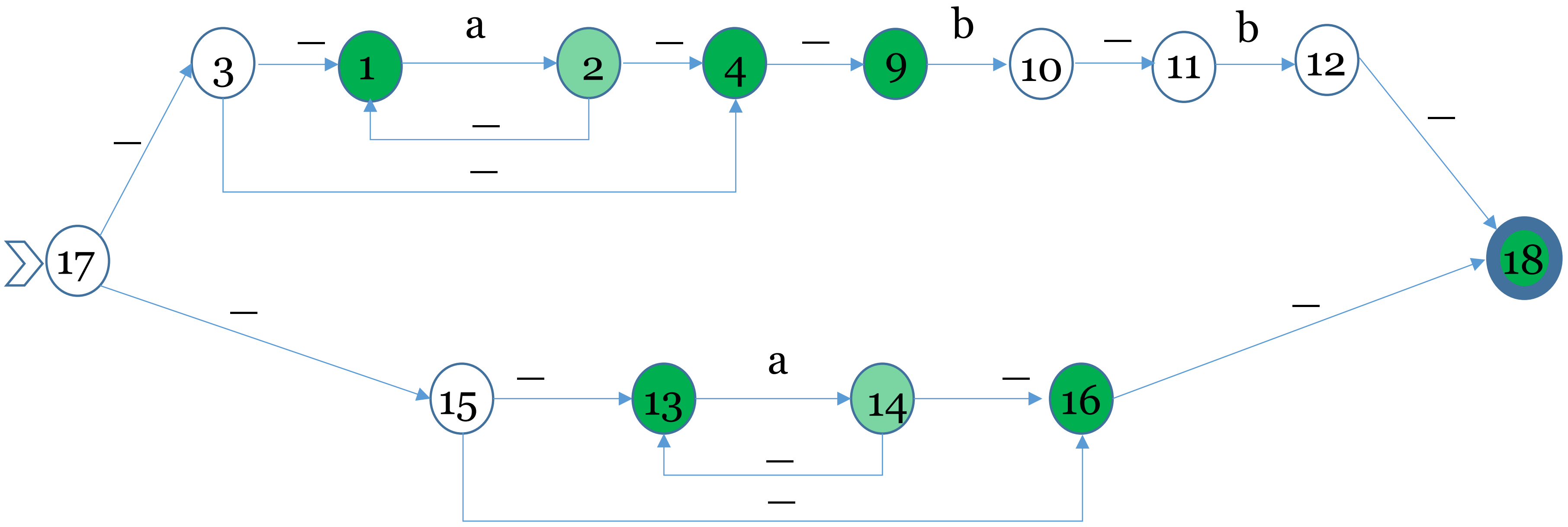
So, effectively, from the initial node 17, we can go to the set { 1 , 2 , 4, 9, 13, 14, 16, **18**} by consuming a. For the rest of the example, I will not add commentary



REG for (((a)*). ((b).(b))) | ((a)*)

$\xrightarrow{\_}$ { 17, 3, 15, 1, 4, 13, 16, 9, **18** } $\xrightarrow{a}$ { 2 , 14 } $\xrightarrow{\_}$ { 1 , 2 , 4, 9, 13, 14, 16, **18**}

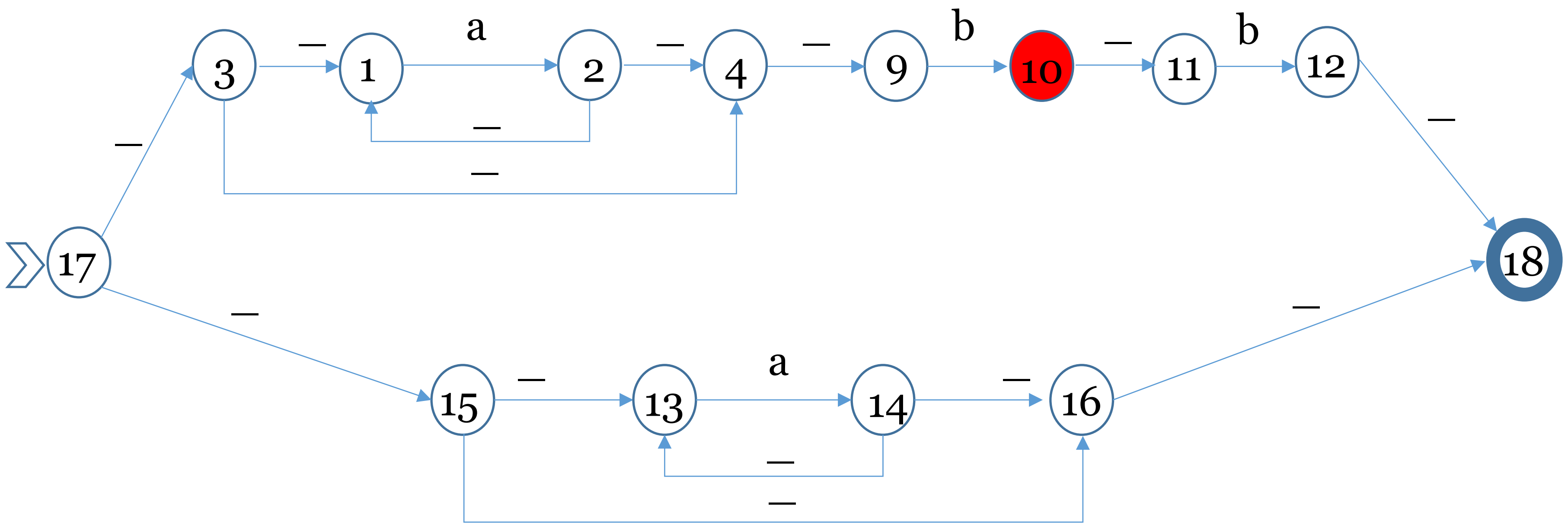
$\xrightarrow{b}$ {



REG for (((a)*). ((b).(b))) | ((a)*)

$\xrightarrow{\_} \{\ 17, 3, 15, 1, 4, 13, 16, 9, \mathbf{18}\ \} \xrightarrow{a} \{\ 2\ , 14\ \} \xrightarrow{\_} \{\ 1\ , 2\ , 4, 9, 13, 14, 16, \mathbf{18}\}$

$\xrightarrow{b} \{\ 10\ \}$

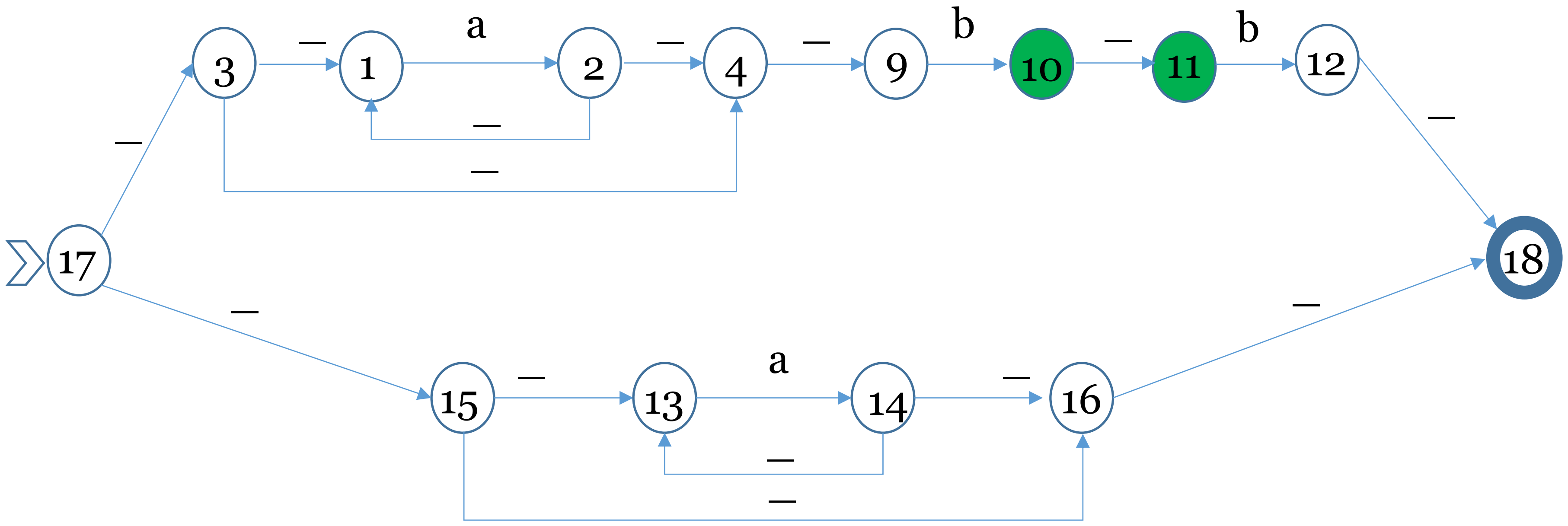


REG for (((a)*). ((b).(b))) | ((a)*)

$\xrightarrow{\_}$ { 17, 3, 15, 1, 4, 13, 16, 9, **18** } $\xrightarrow{a}$ { 2 , 14 } $\xrightarrow{\_}$ { 1 , 2 , 4, 9, 13, 14, 16, **18**}

$\xrightarrow{b}$ { 10 } $\xrightarrow{\_}$ { 10 , 11 }

So far, ab is viable but not matching



REG for (((a)*). ((b).(b))) | ((a)*)

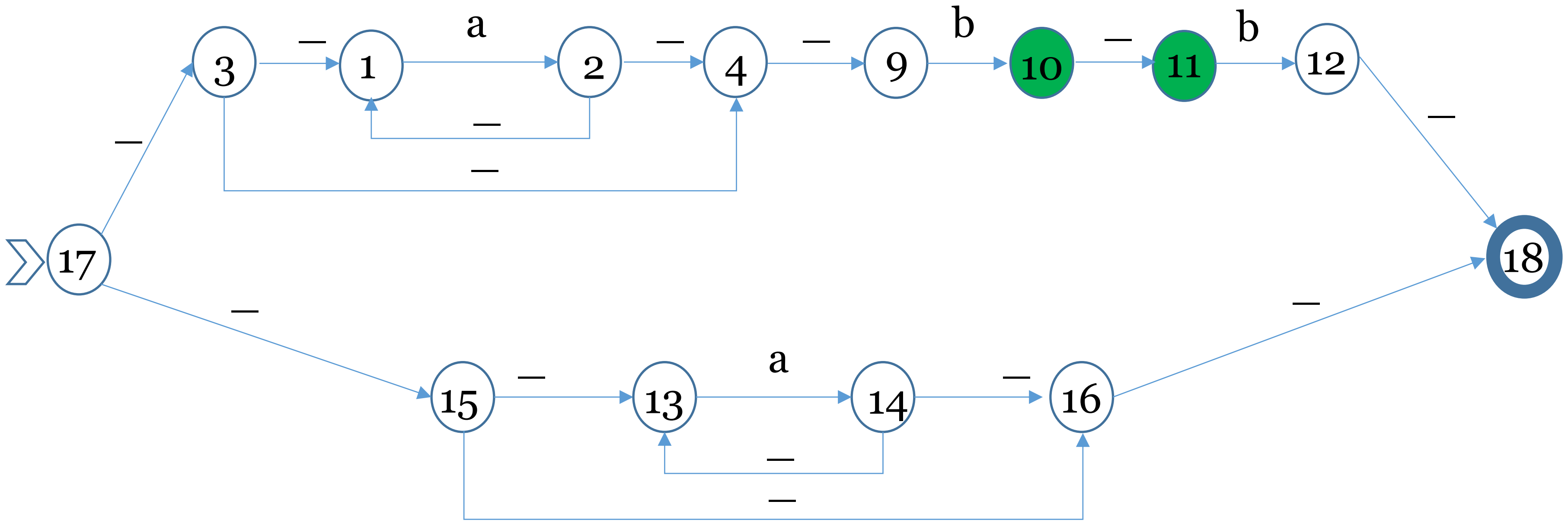
$\xrightarrow{\_}$ { 17, 3, 15, 1, 4, 13, 16, 9, **18** } $\xrightarrow{a}$ { 2 , 14 } $\xrightarrow{\_}$ { 1 , 2 , 4, 9, 13, 14, 16, **18**}

$\xrightarrow{b}$ { 10 } $\xrightarrow{\_}$ { 10 , 11 }

So far, ab is viable but not matching.

Not matching means that the set of nodes that can be reached by consuming ab does not contain the accept node

viable means that the set that can be reached by consuming ab is not empty, so there is still hope to reach the accepting state by consuming characters yet to come.

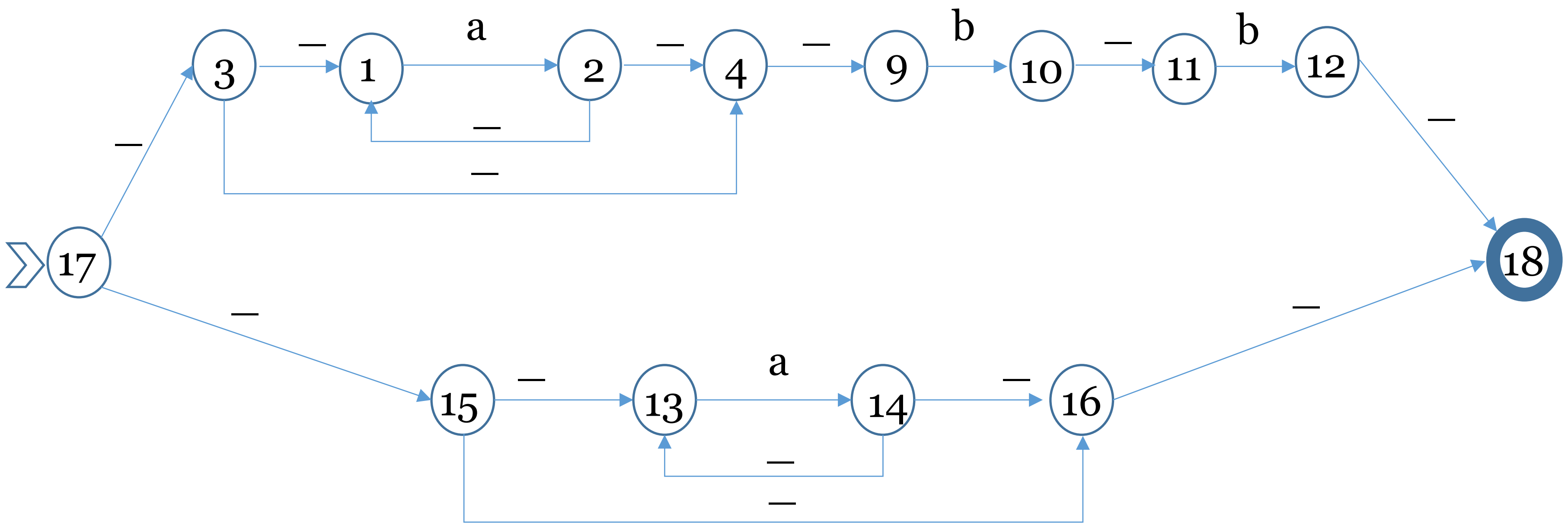


REG for (((a)*). ((b).(b))) | ((a)*)

$\xrightarrow{-}$ { 17, 3, 15, 1, 4, 13, 16, 9, **18** } $\xrightarrow{a}$ { 2 , 14 } $\xrightarrow{-}$ { 1 , 2 , 4, 9, 13, 14, 16, **18**}

$\xrightarrow{b}$ { 10 } $\xrightarrow{-}$ { 10 , 11 } $\xrightarrow{a}$ {   }

aba is not viable nor matching because no nodes can be reached by consuming aba



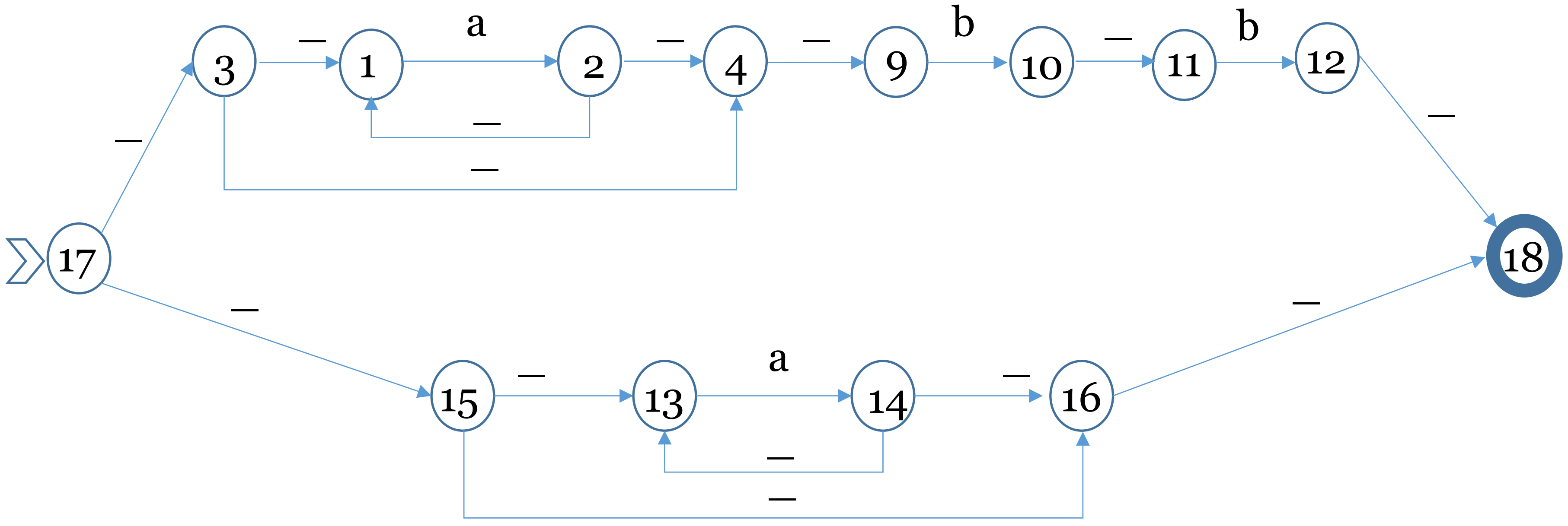
REG for (((a)*). ((b).(b))) | ((a)*)

Examples of reachable nodes

$\xrightarrow{\_}$ { 17, 3, 15, 1, 4, 13, 16, 9, **18** } $\xrightarrow{a}$ { 2 , 14 } $\xrightarrow{\_}$ { 1 , 2 , 4, 9, 13, 14, 16, **18**}

$\xrightarrow{b}$ { 10 } $\xrightarrow{\_}$ { 10 , 11 } $\xrightarrow{a}$ { }

"a" is returned and next call will start after the "a" and returns ERROR



REG for (((a)*). ((b).(b))) | ((a)*)

8. How is restated main theorem relevant?

Recall our task

- Given:
    - a REG *r* for regular expression *R*,
    - a string *s*, and
    - a position *p* in the string,

- Find:
    - longest possible substring *w* starting at *p* such that ***w* is in L(*R*)**

How is main theorem relevant?

- Given:
  - a REG *r* for regular expression *R*,
  - a string *s*, and
  - a position *p* in the string,

- Find:

  longest possible substring *w* starting at *p* such that **the accepting state of *r* is reachable by consuming *w***