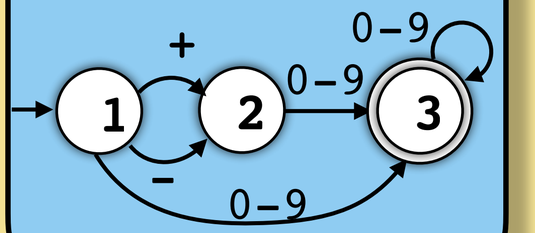
Lexical Analysis 2- Finite State Machines

```
foo(){
   ... ... ...
}
```

Compiler

Lexical Analysis



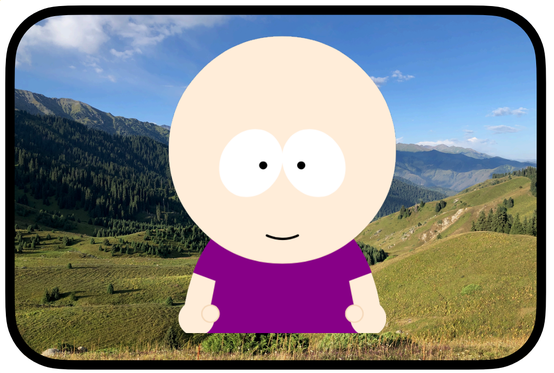
Syntactic Analysis

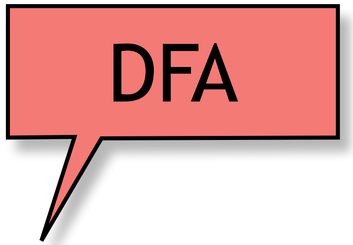
Semantic Analysis

Intermediate Code Generation

Code Optimization

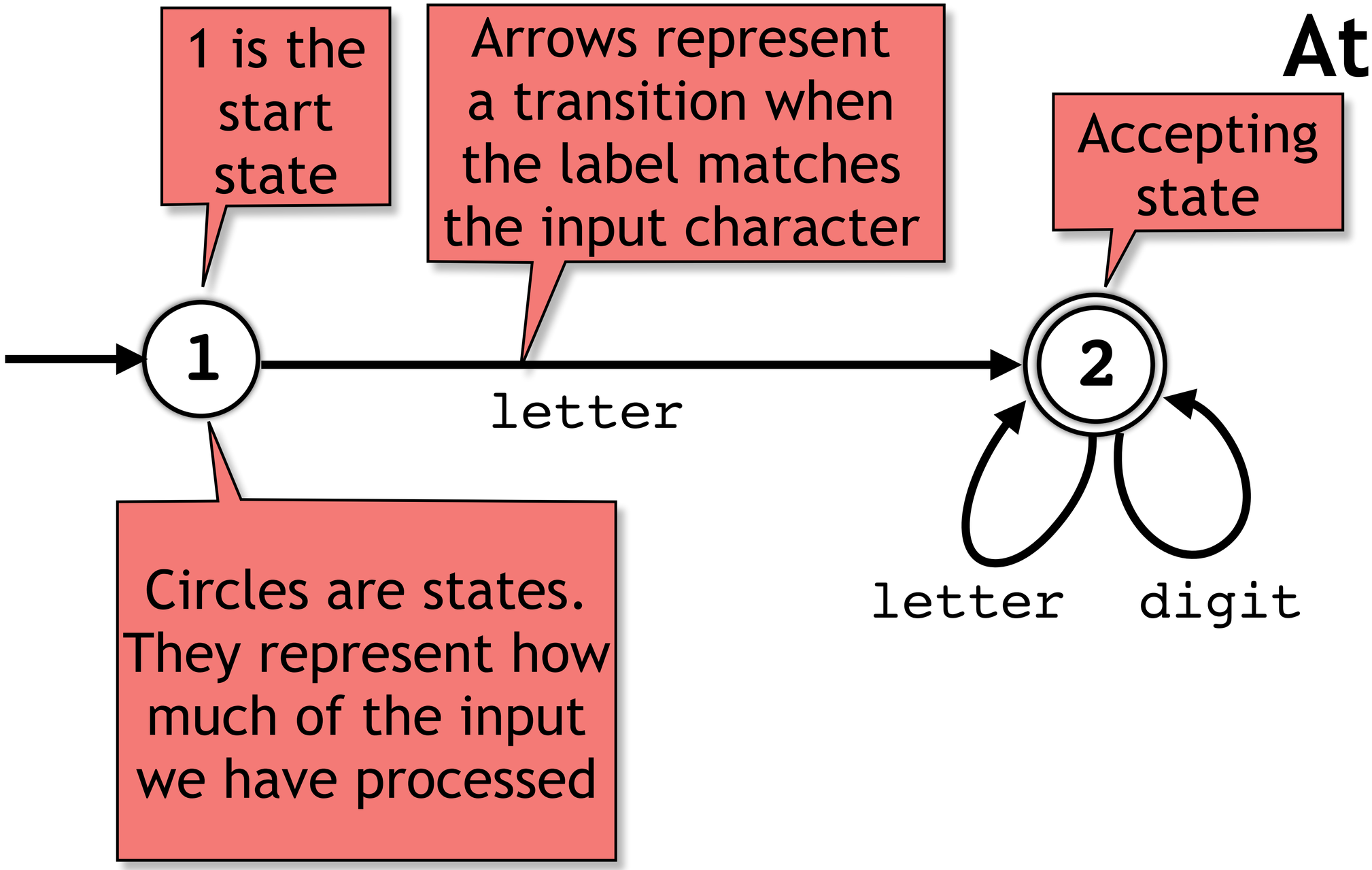
Machine Code Generation



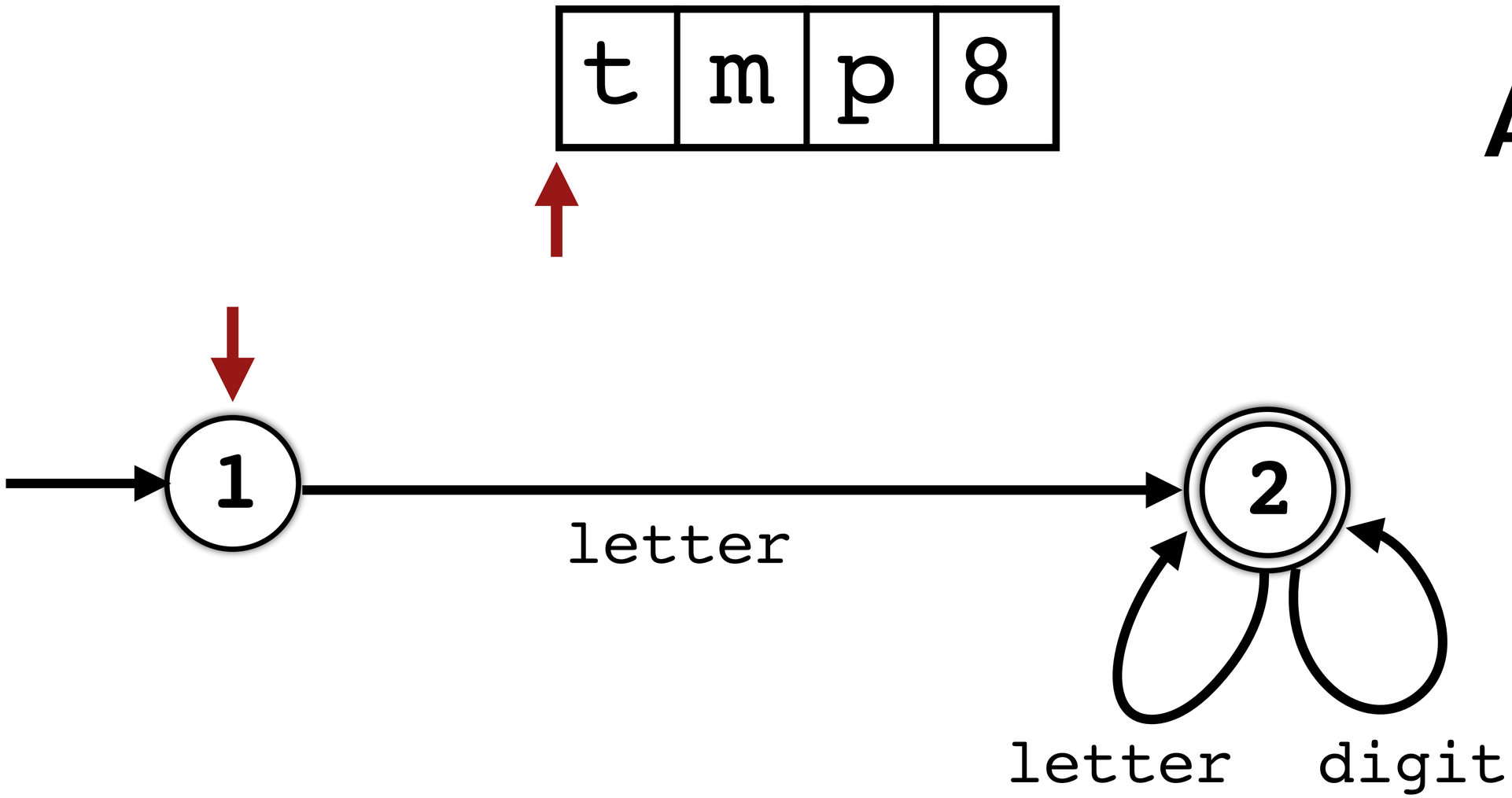


Deterministic Finite Atomaton

Finite Atomata



Finite Atomata



t m p 8

1 → 2 → 2 → 2 → 2

Deterministic Finite Automaton

A Deterministic Finite [illegible] M c

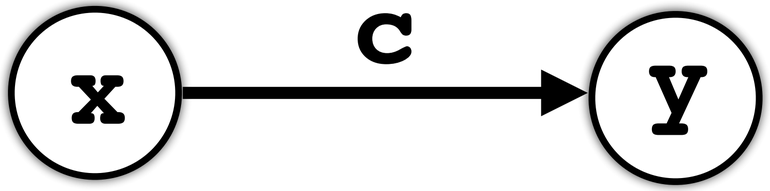
- An alphabet Σ
- A set of states S,
- A transition function $T : S \times \Sigma \rightarrow S$,
- A start state $s_0 \in S$,
- A set of accepting states $A \subset S$.

Take a state …

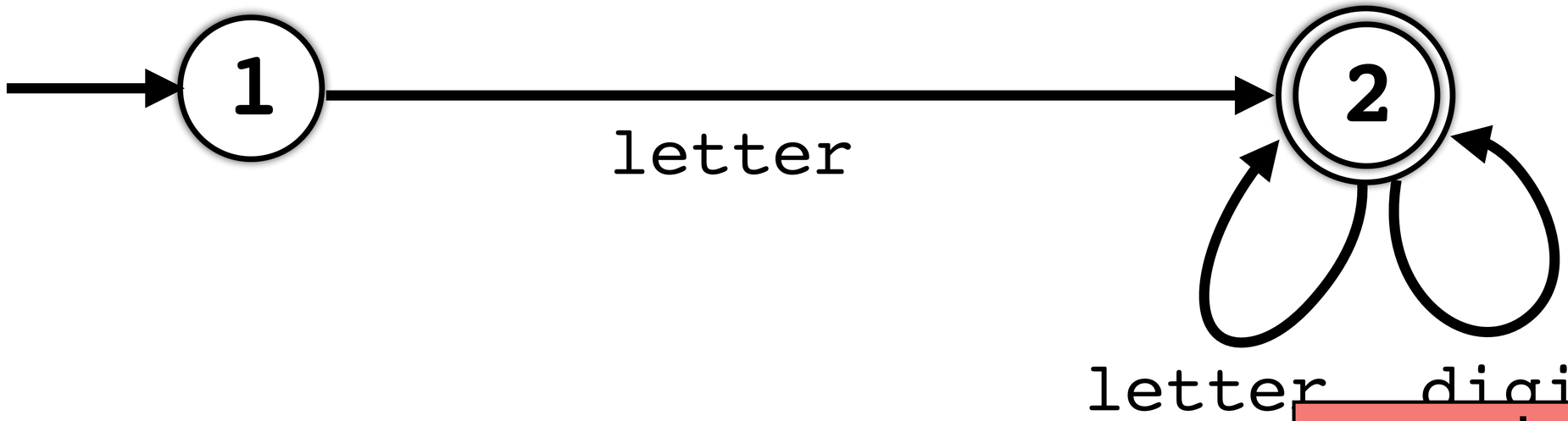
… and an input symbol …

… and get a new state

T records the transitions between states, depending on input:



T(x, c) = y



- An alphabet Σ = **letters and digits**
- A set of states S = **{1,2}**
- A start state s_0 = **1**
- A set of accepting states A = **{2}**

- A transition function **T(x, c)=**

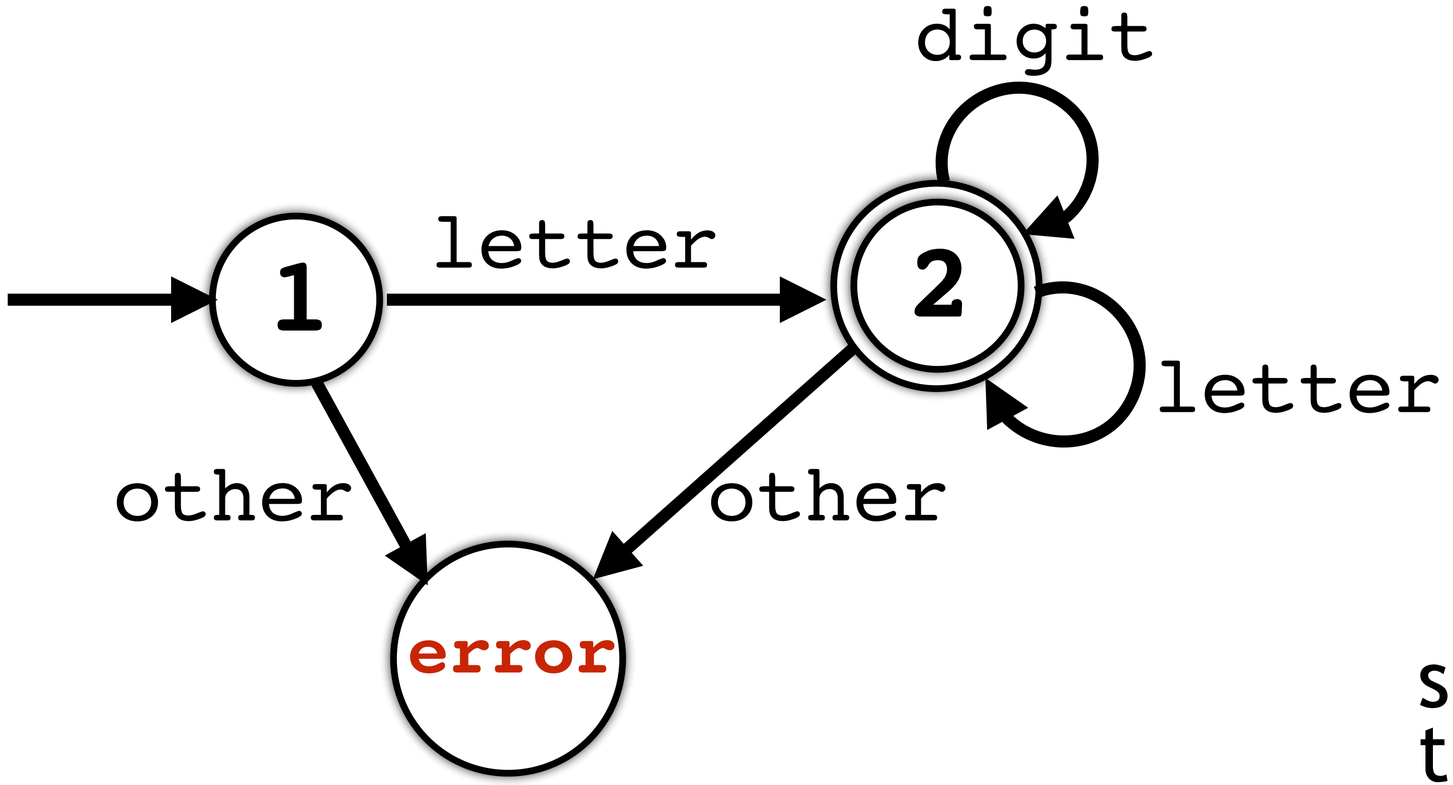
state \ input	letter	digit
1	2	
2	2	

Take a state ...

... and an input symbol ...

... and get a new state

Error states



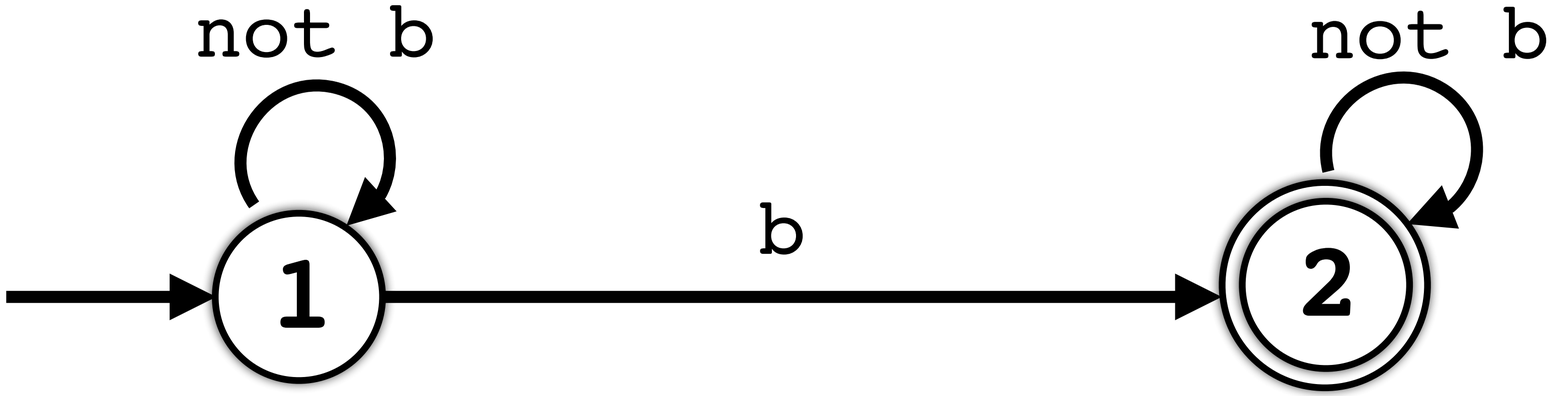
$T(x, c)$ =

state	letter	digit
1	2	error
2	2	2

input

Exercises

Strings with exactly one b



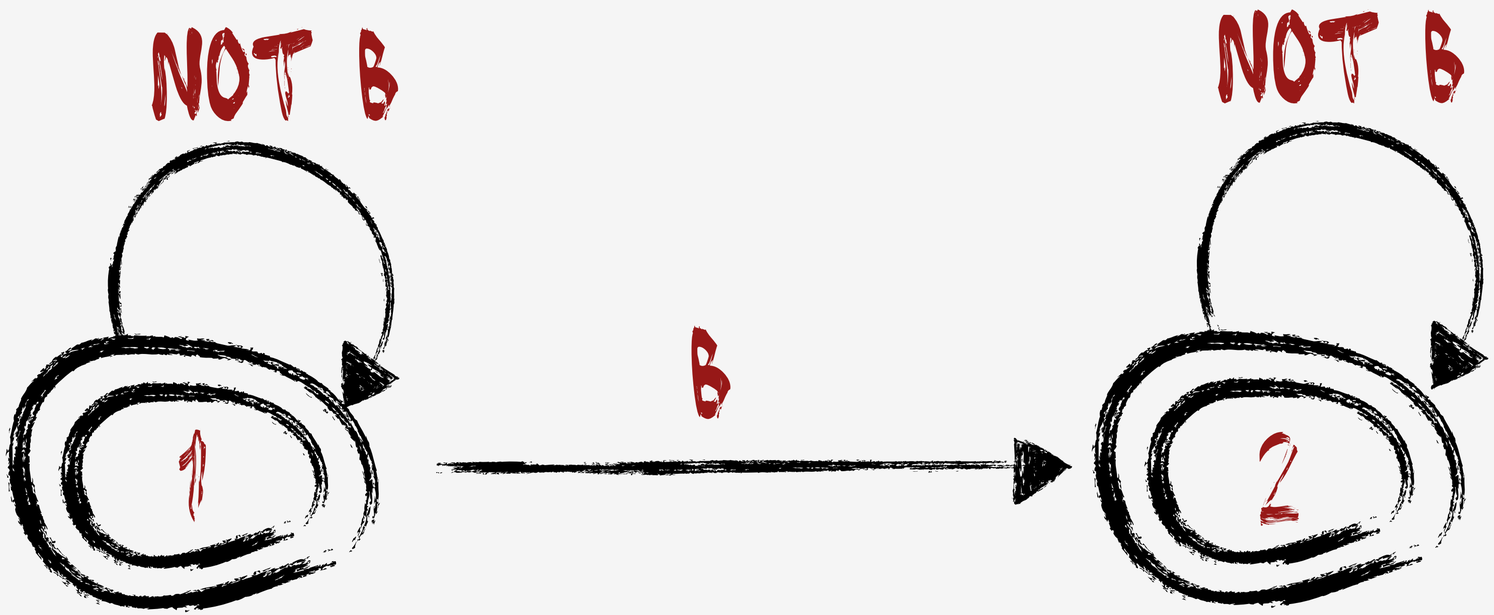
- b
- ab
- aab
- aabaa
- aaaaaab

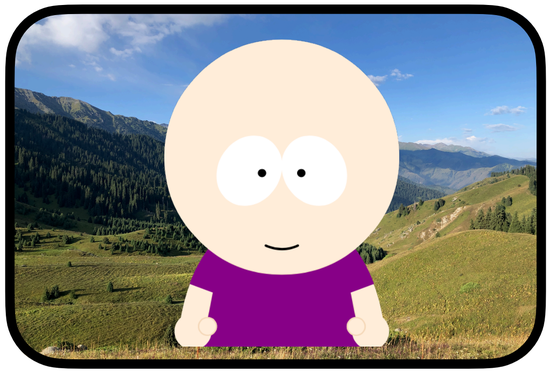
- ''
- a
- abb
- abab

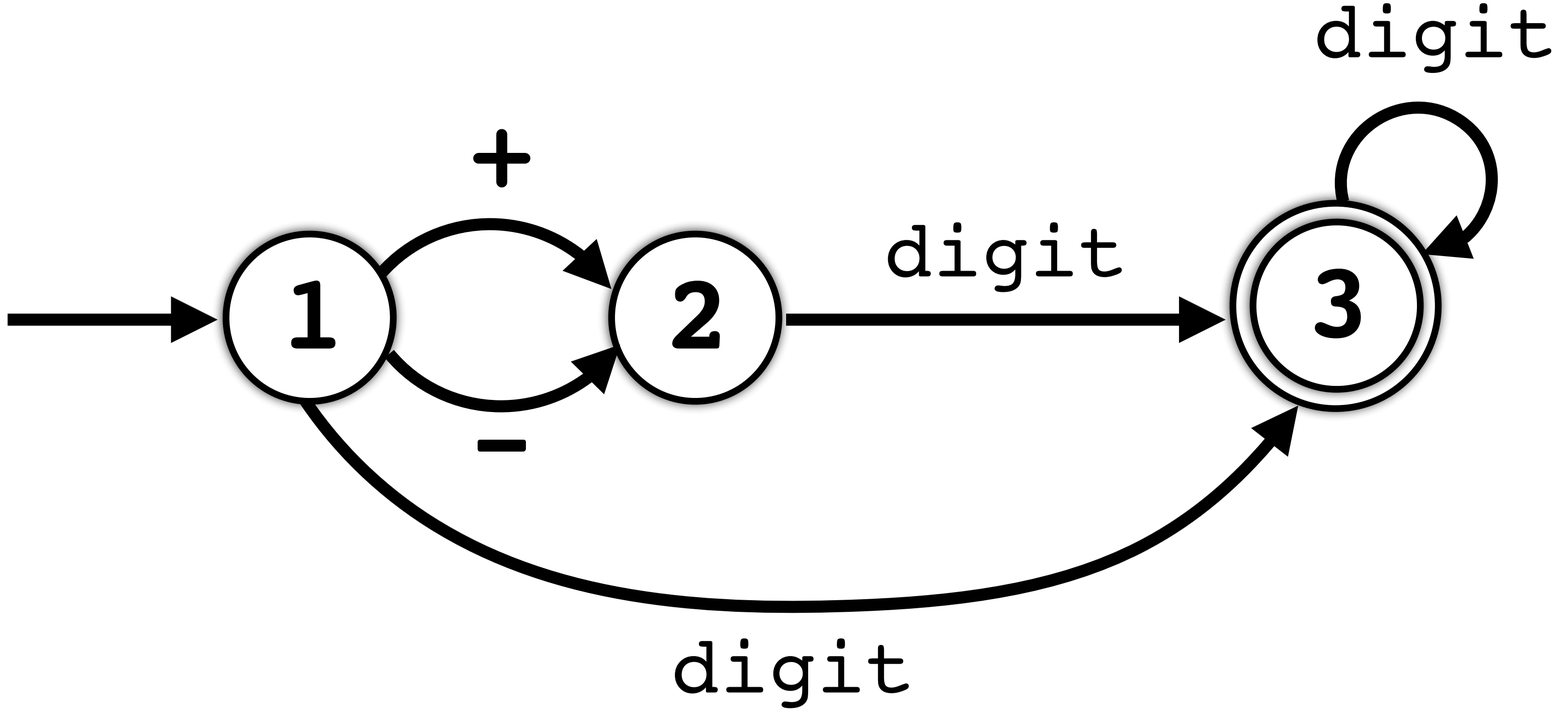
Strings with at most one (0 or 1) b

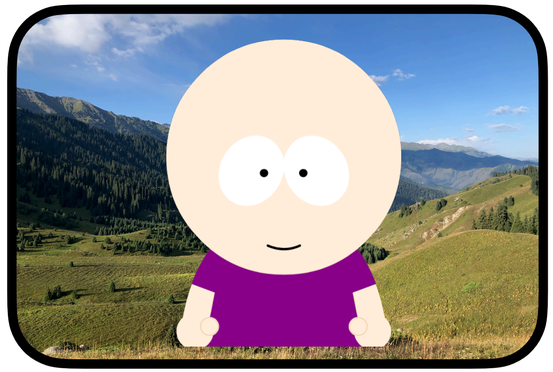
- ''
- b
- a
- aba
- aabaa
- aaaaaa

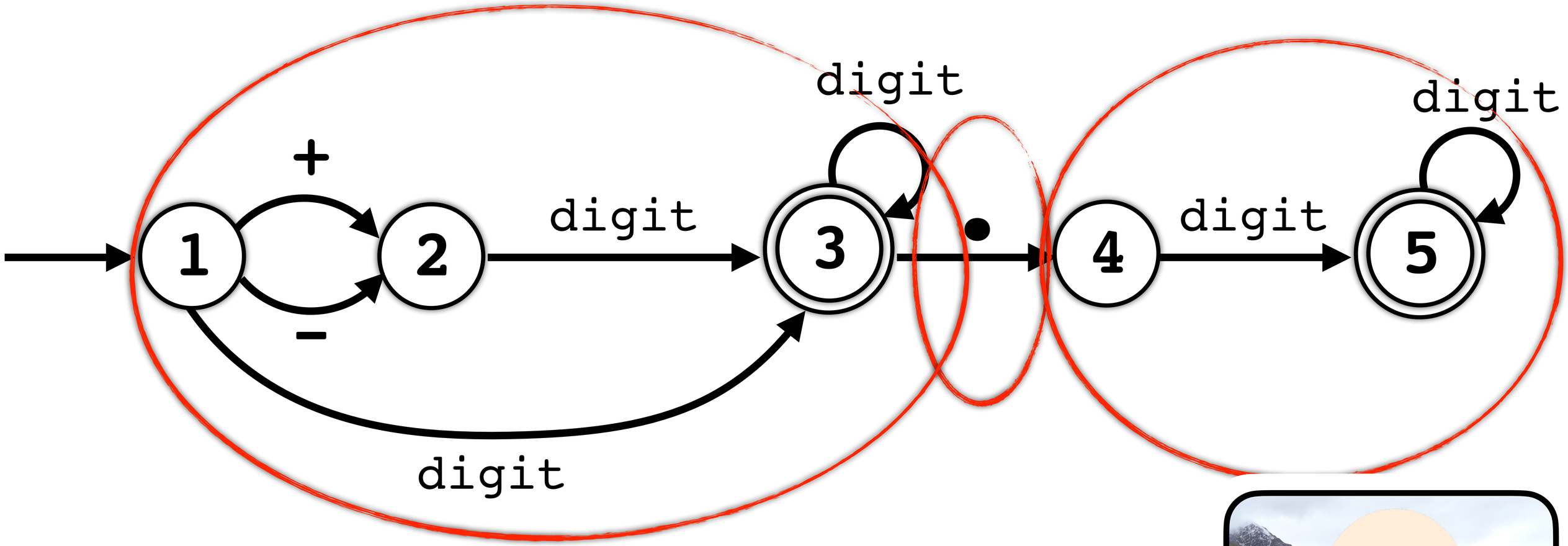
- abb
- abab

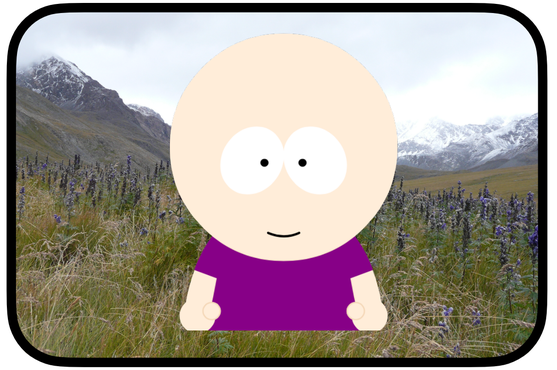

NOT B
1
B
NOT B
2



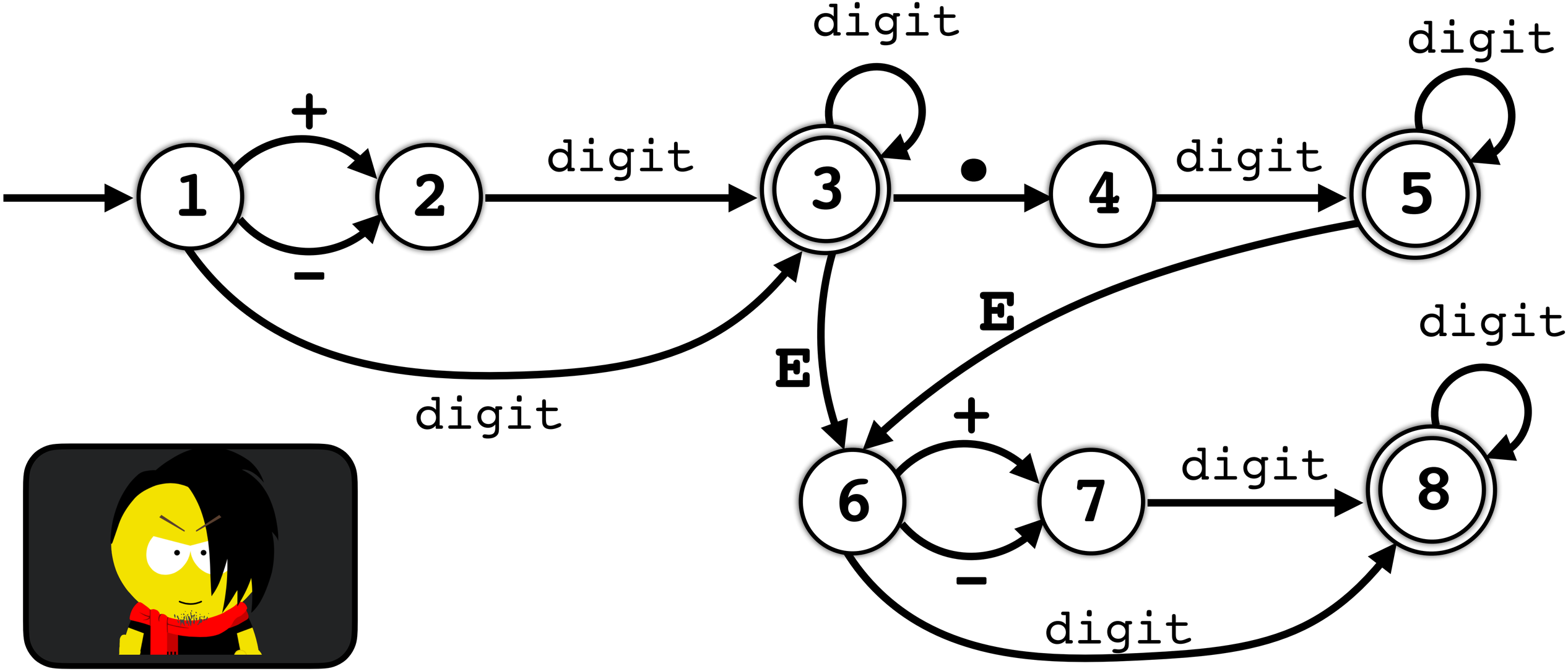

digit
+
digit
1
2
3
−
digit



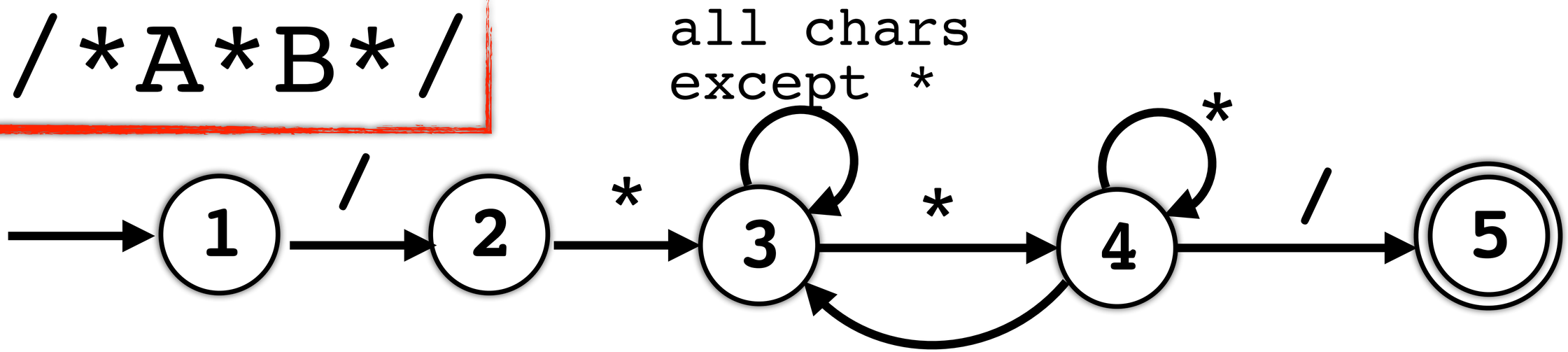

digit
digit
digit
+
−
1
2
3
4
5
.
digit



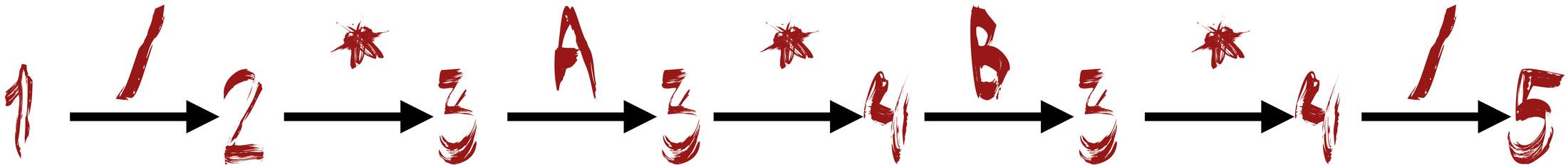
+8675309 1.618033 6.022E23



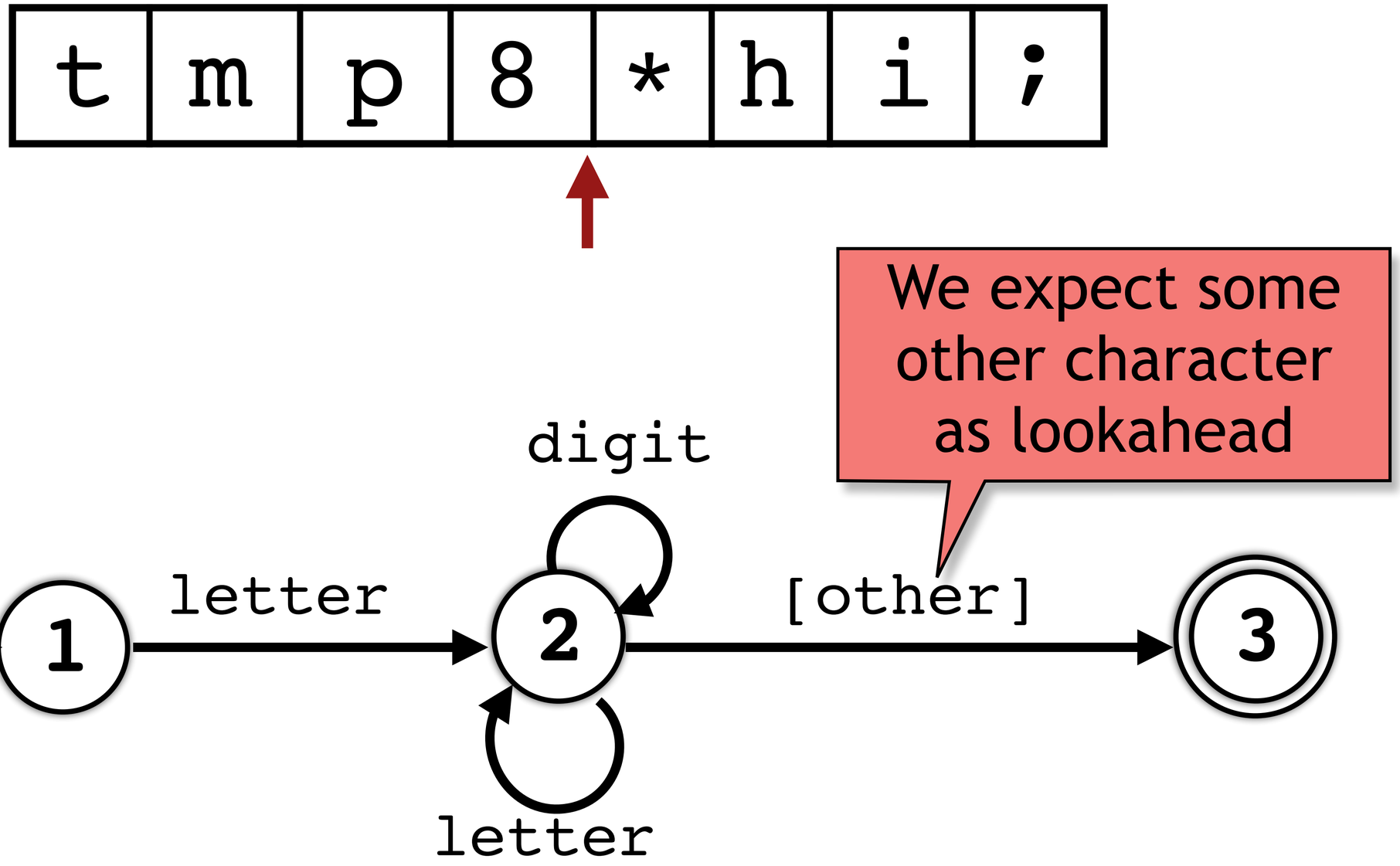
/*A*B*/



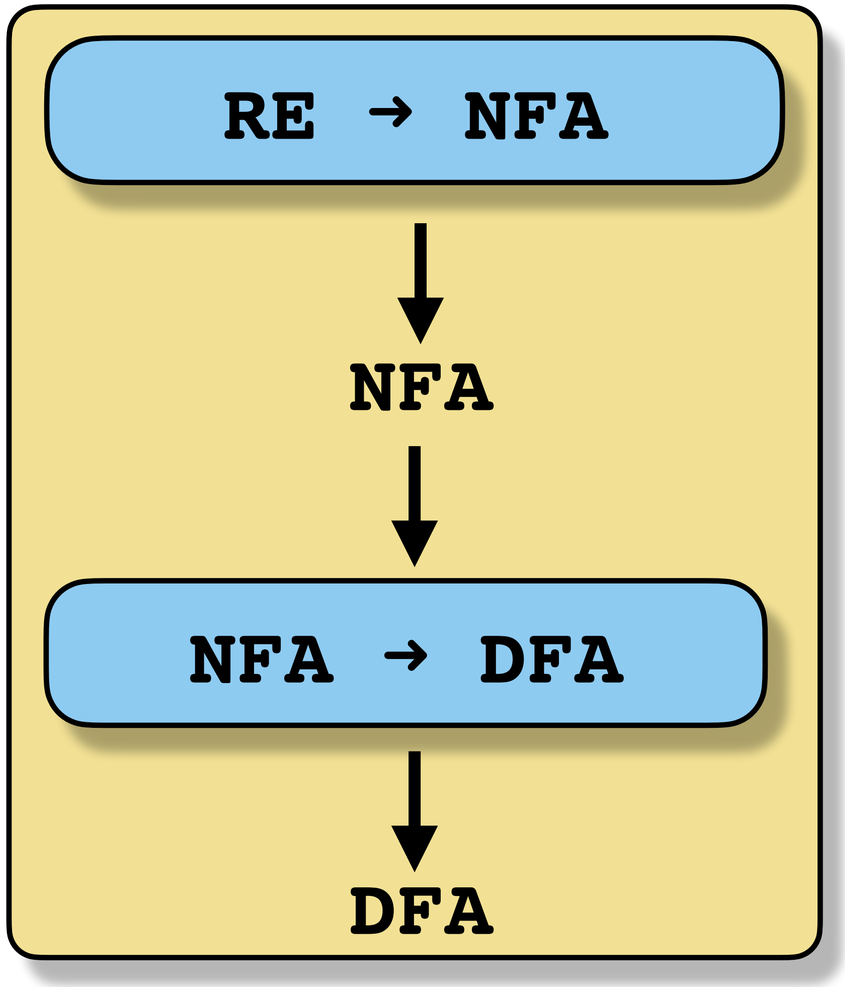
all chars except * and /



Lookahead


t m p 8 * h i ;
We expect some other character as lookahead
digit
letter
[other]
1
2
3
letter

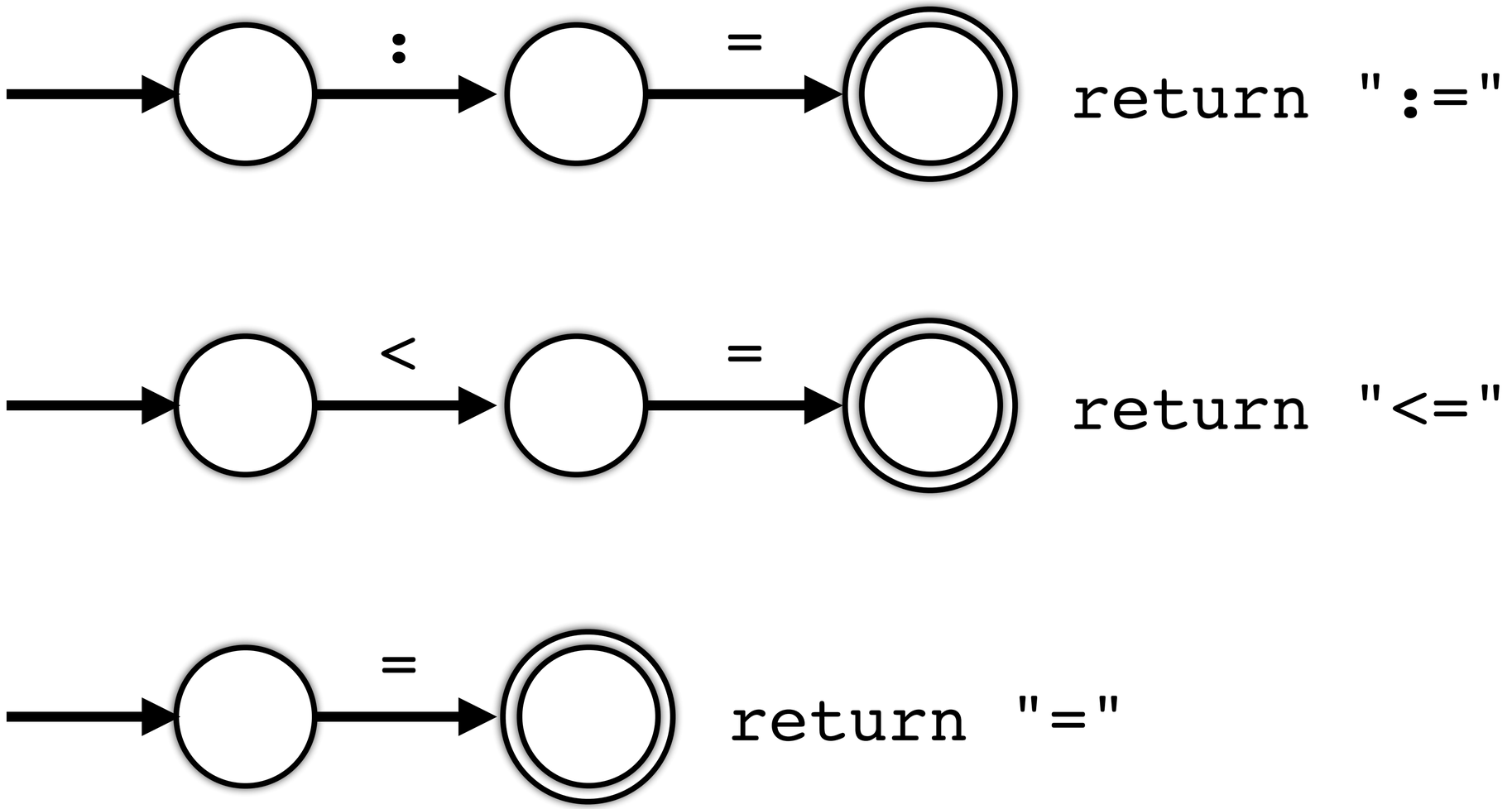
Regular Expression



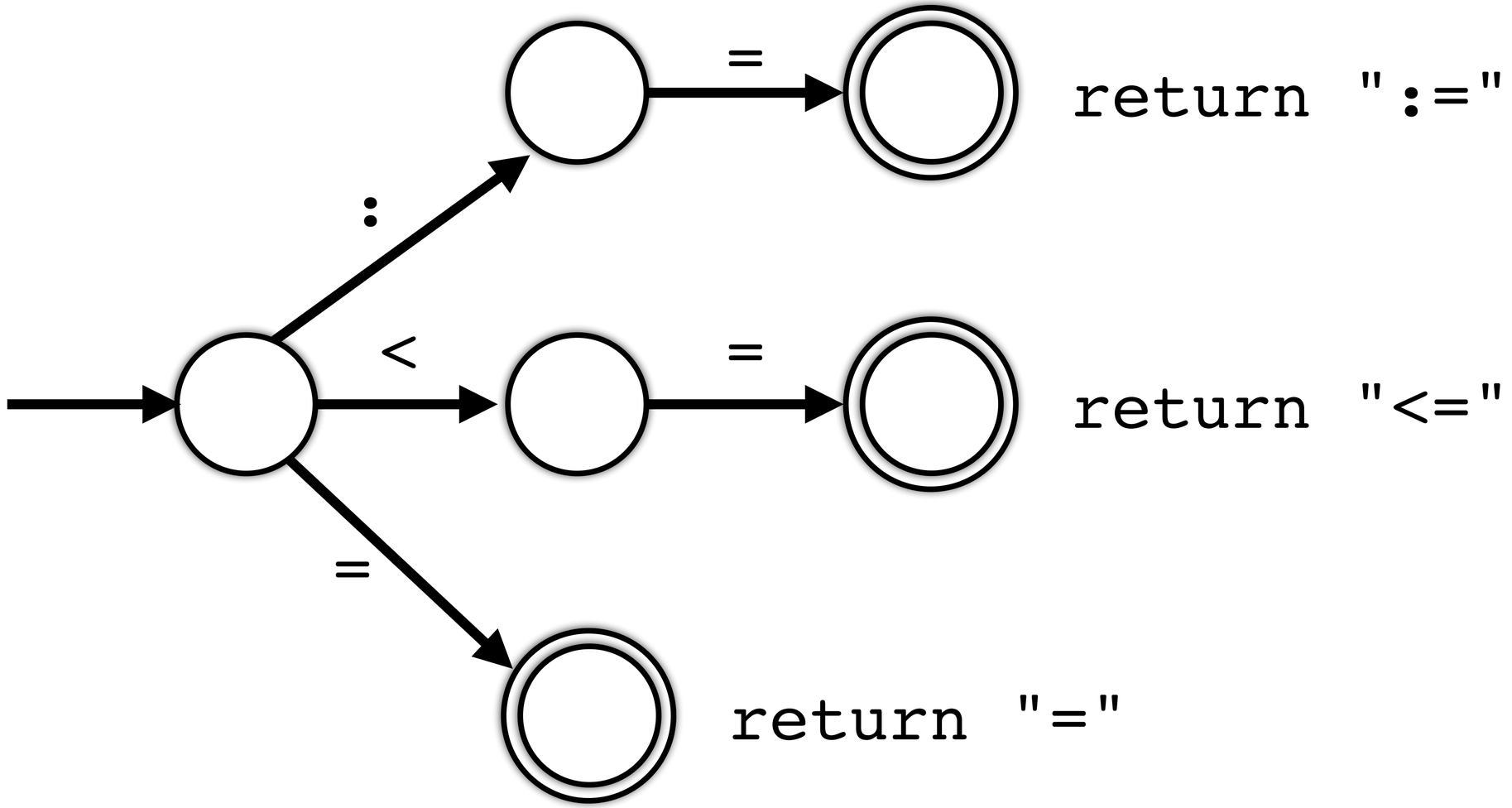

```
public static boolean interpret() {
   int state = 0;
   int ch = nextChar();
   while(true) {
      switch (state) {
         case -1 : return false;
         case 0 : { }
         case 1 : { }
         case 2 : { }
         case 3 : { }
         case 4 : return (ch==END);
      }
   }
}
```

Nondeterministic Finite Atomaton

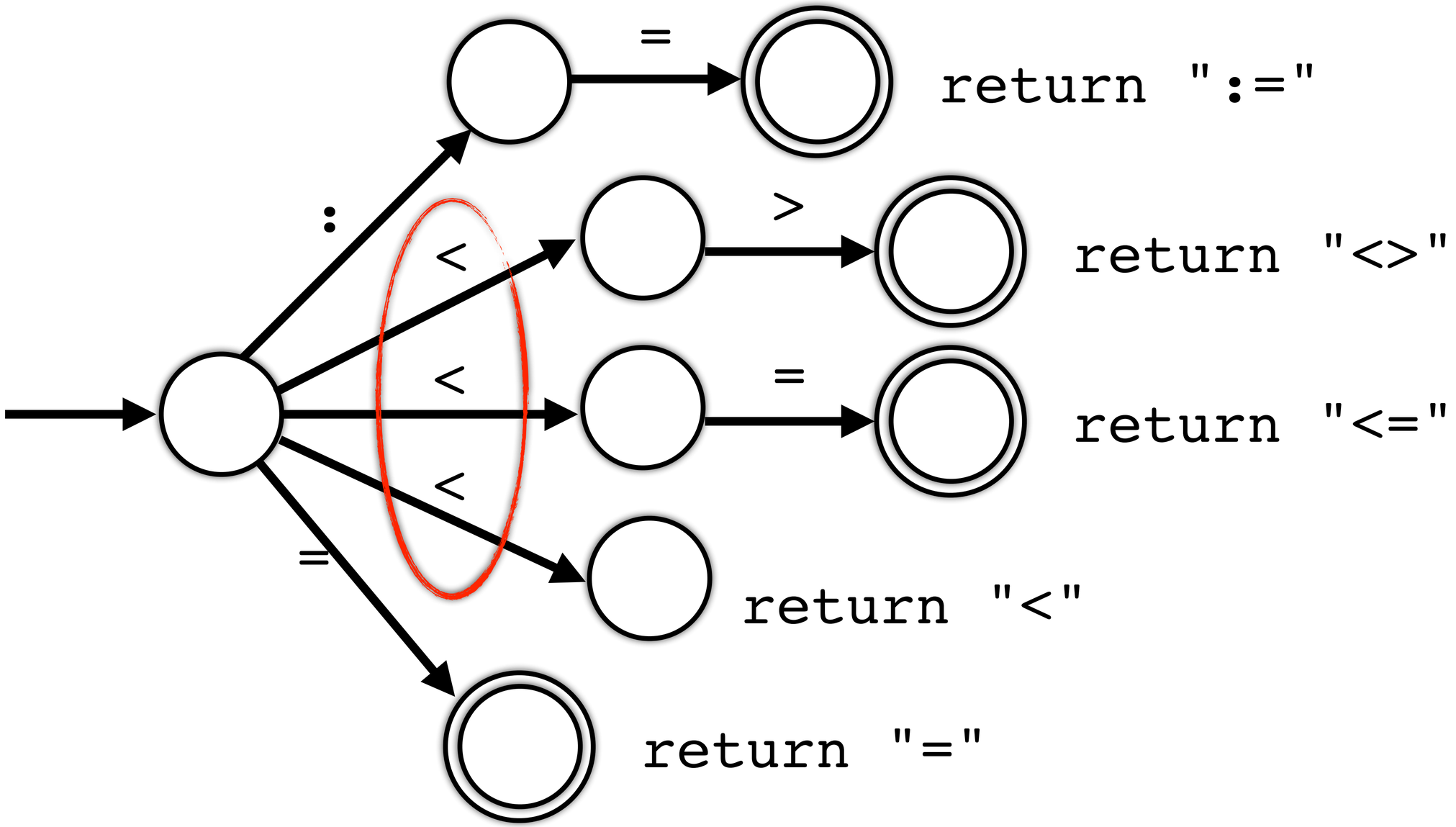
Recognize ":=", "<=", "="



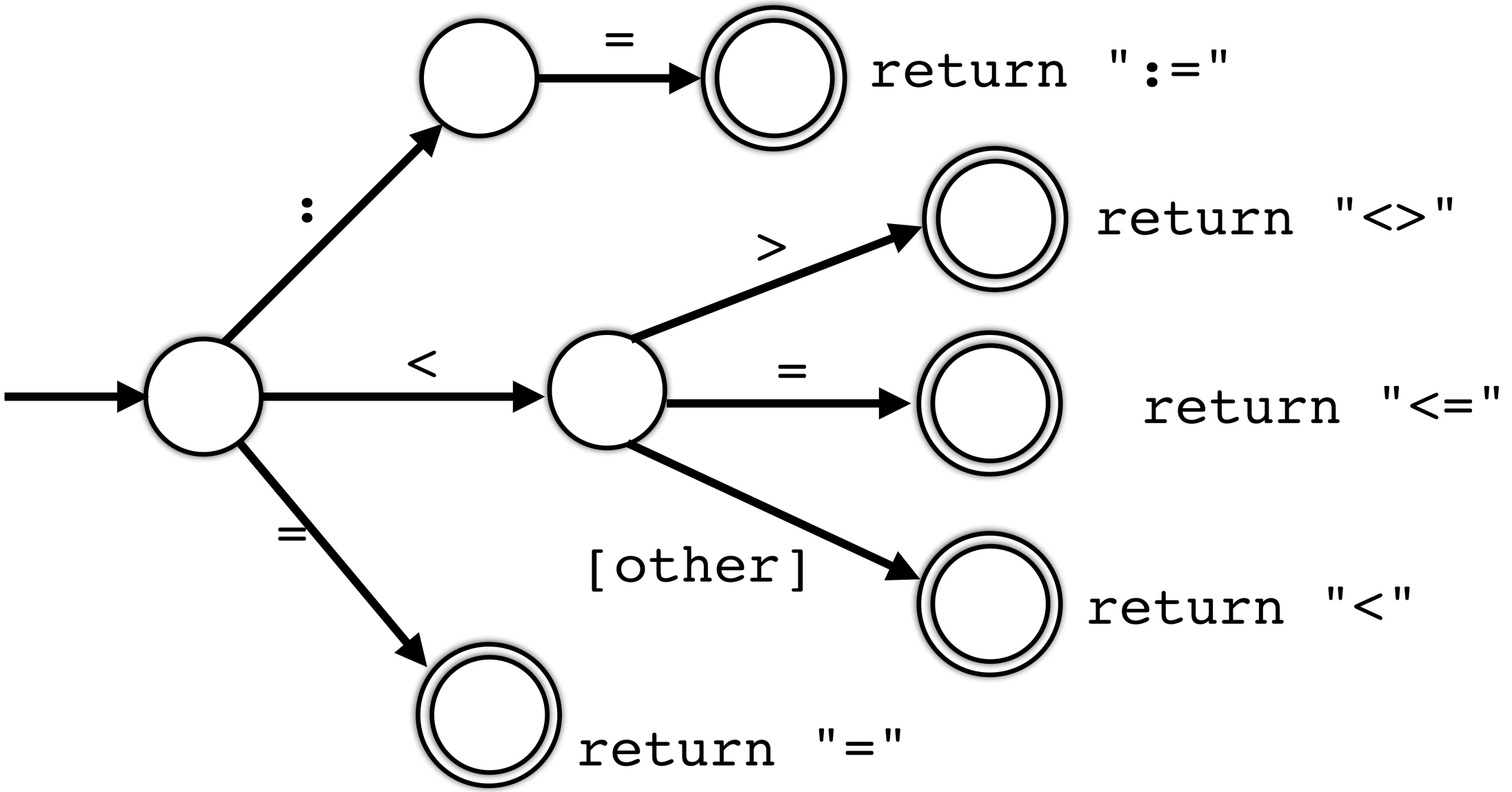
Recognize ":=", "<=", "="



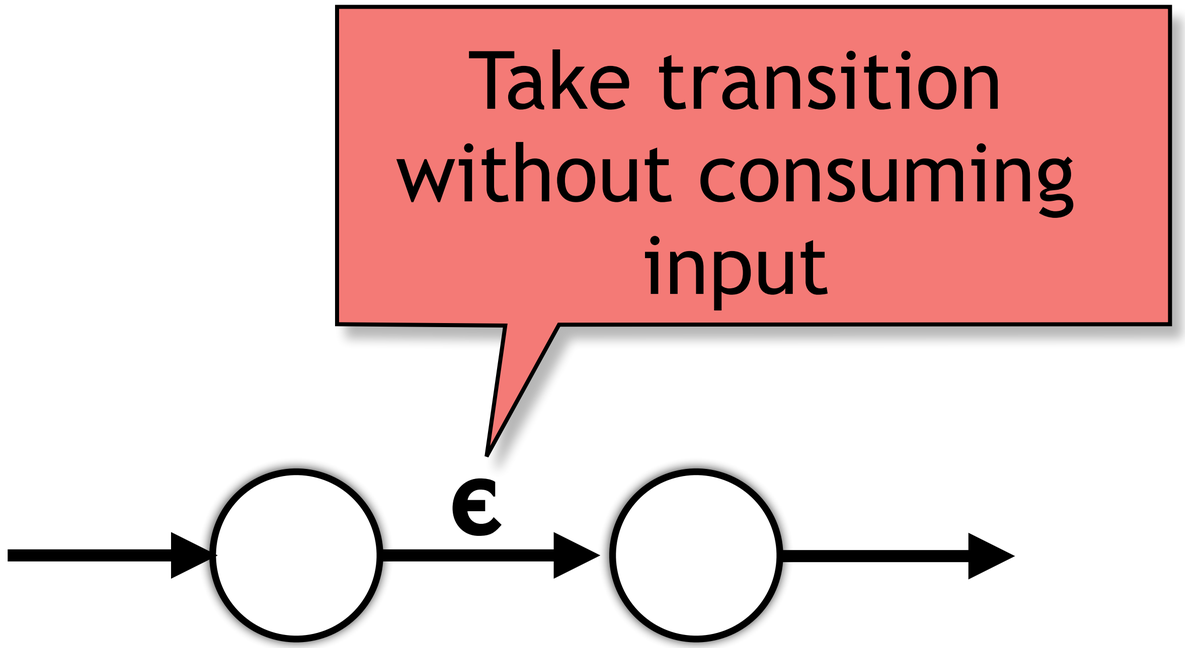
Recognize ":=", "<=", "=", "<", "<>"



Recognize ":=", "<=", "=", "<", "<>"



Epsilon Transitions



Nondeterministic Finite Automaton

A Nondeterministic F ton M (

- An alphabet
- A set of state
- A start state
- A set of accepting states $A \subset S$.
- A transition function $T : S \times (\Sigma \cup \{\epsilon\}) \to \mathbb{P}(S)$.

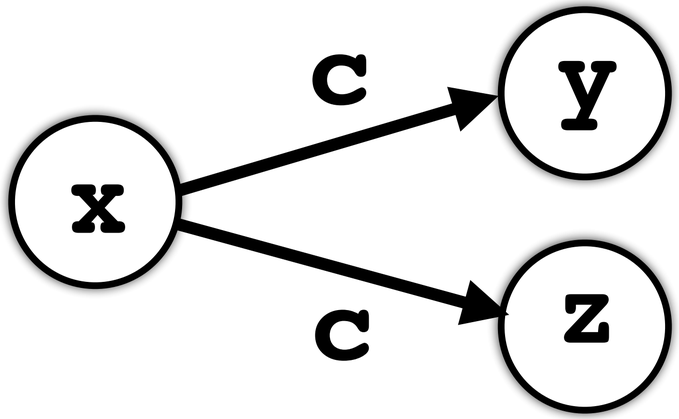
Take a state ...

... and an input symbol or an ϵ ...

... and move to a new set of states

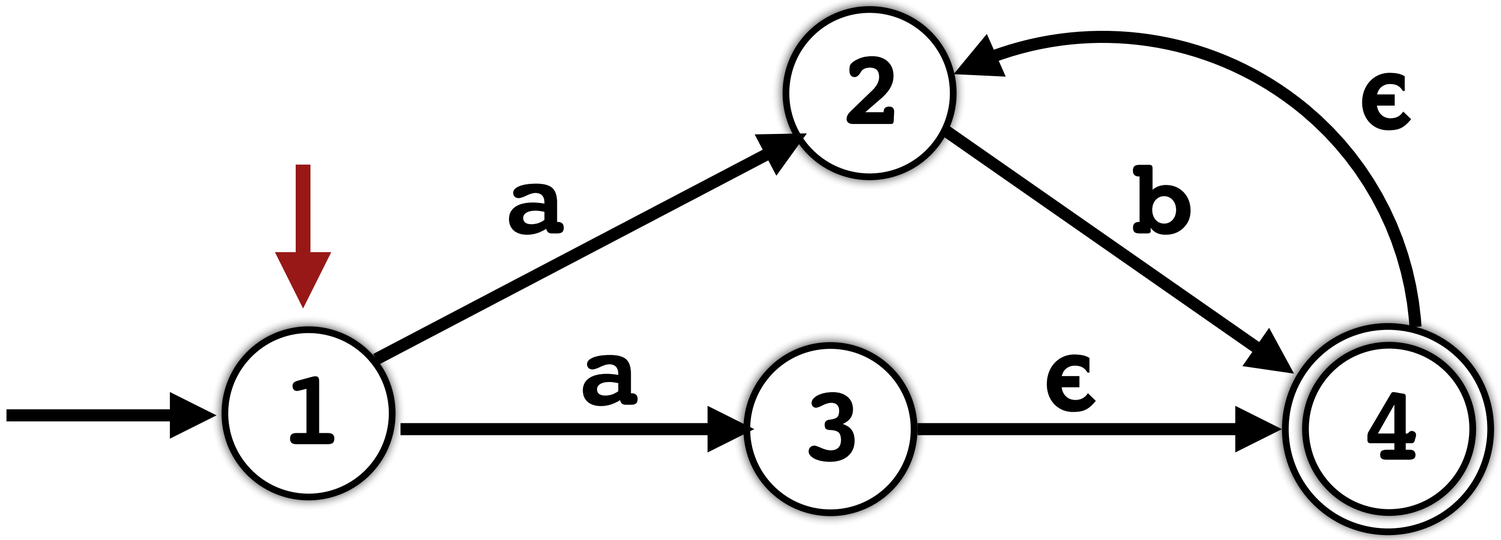
$\mathbb{P}(S)$ is the power-set of S, the set of all subsets

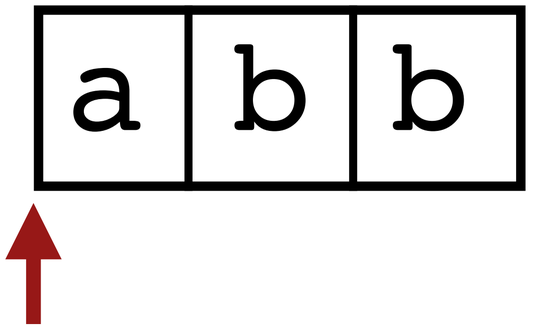
On any transition, we can go to a set of states:

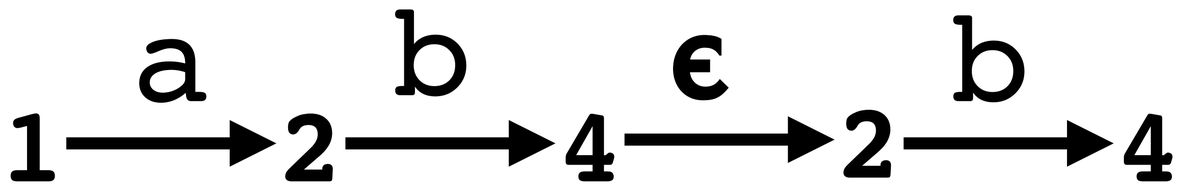


T(x, c) = {y,z}

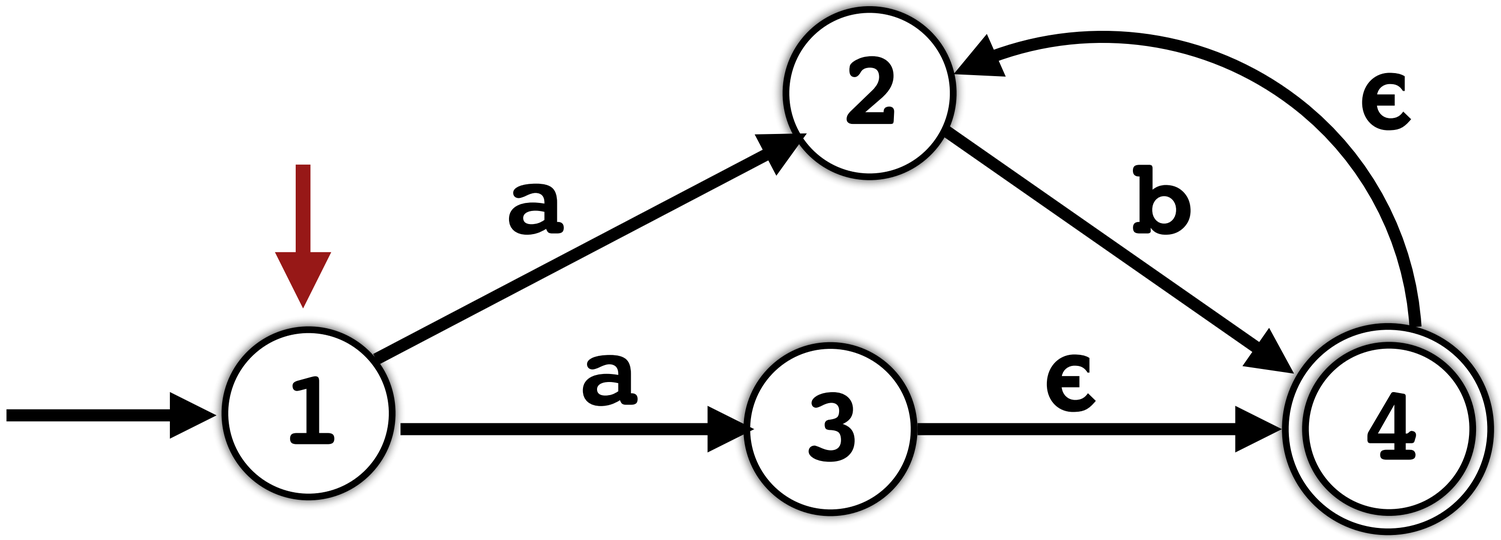
NFA Example

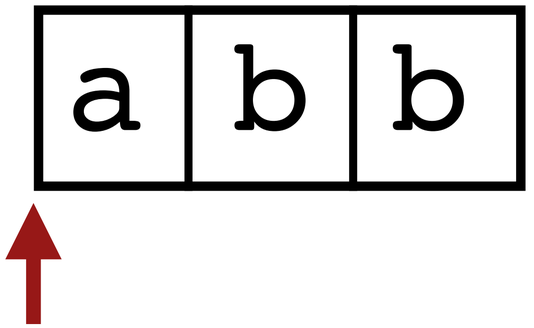


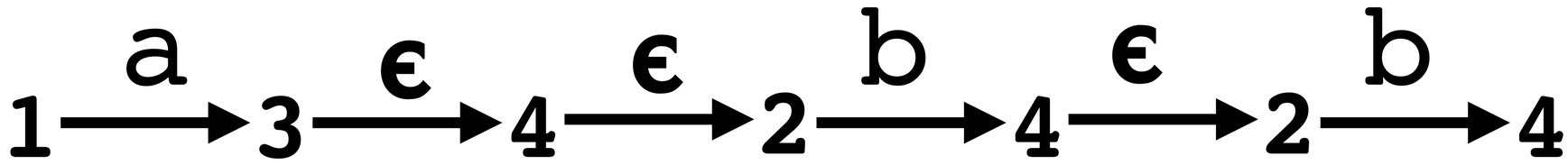




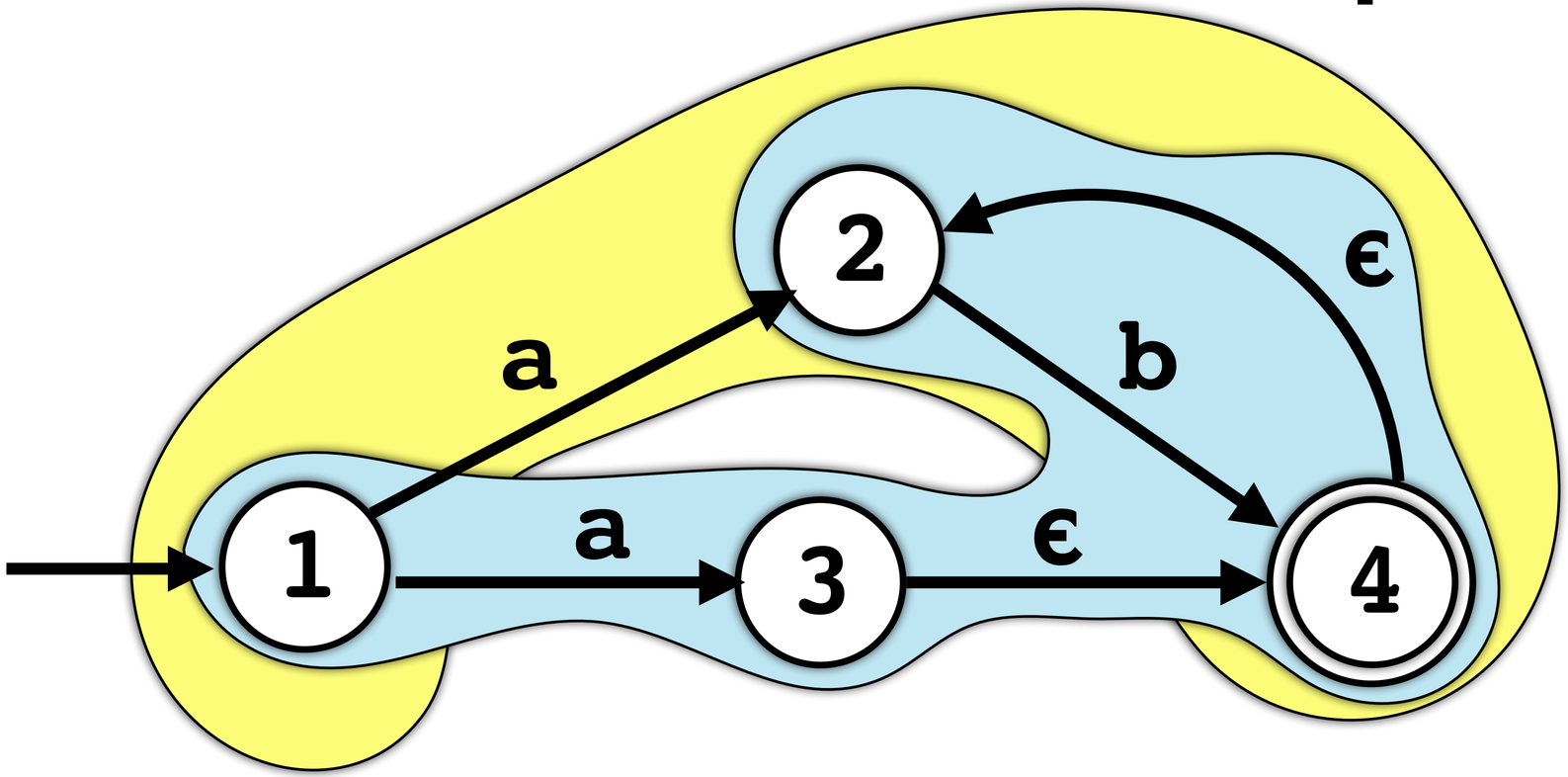
NFA Example





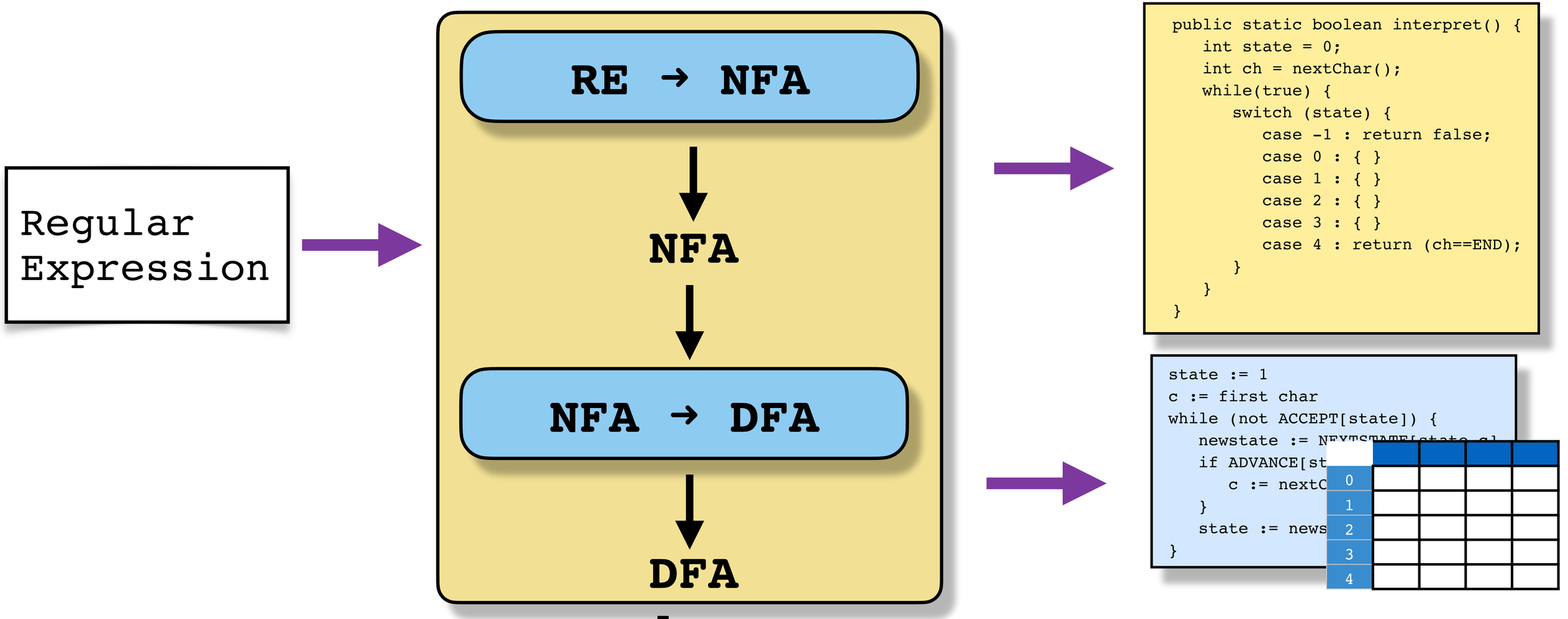


NFA Example



The NFA accepts: ab+|ab*

Or, simpler: ab*



Implementing Lexical Analysis

```
state := start state
c := first char
while (true) {
    switch (state) {
        case state_1: {
            switch (c) {
                case char1 : {
                    c := nextChar();
                    state := new state;
                }
            }
        }
        case state_2: {
            ...
        }
        ...
    }
}
```

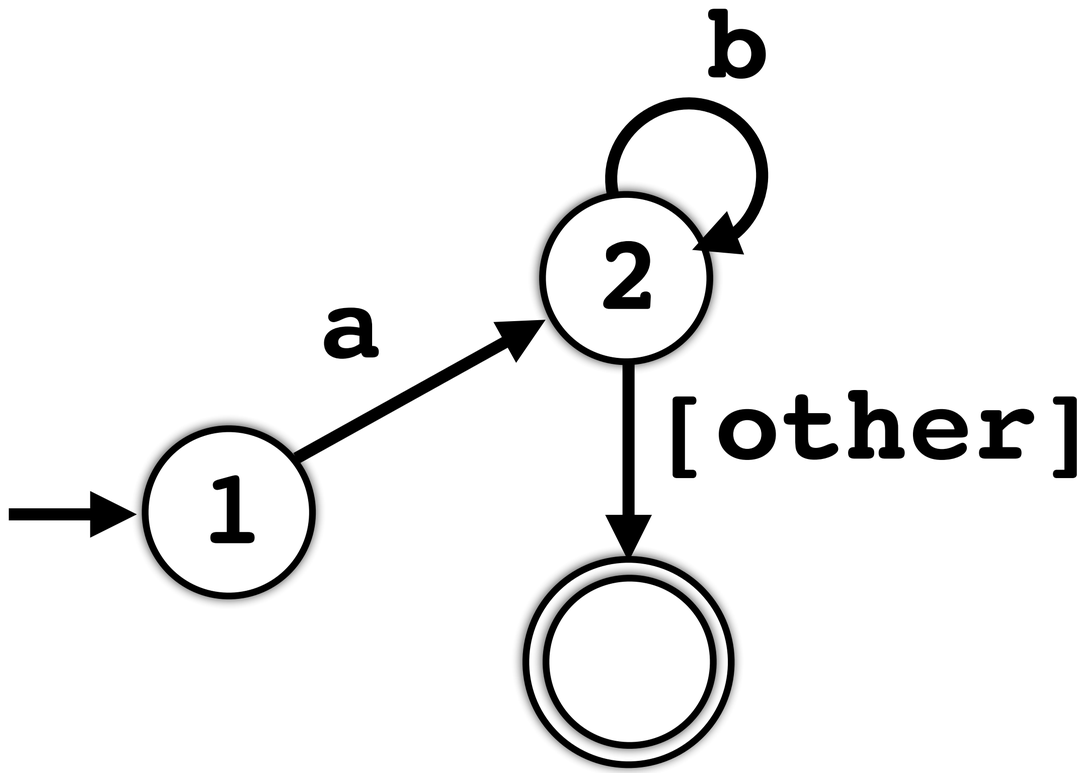
Set the start state and get the first

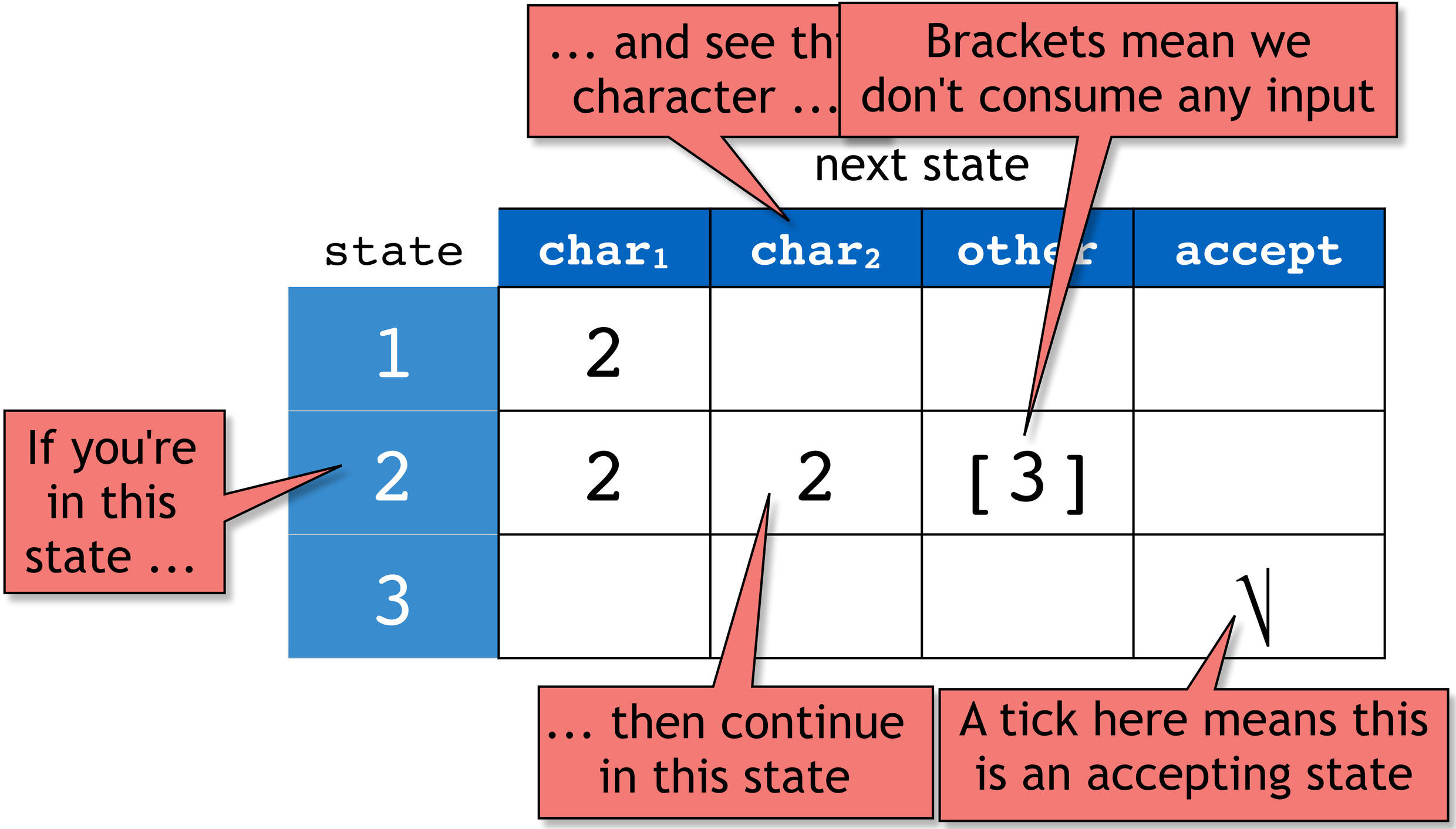
Switch over the current state ...

... and the current character

Move to the next character and select a new state

```
state := state_1
c := first char
while (true) {
    switch (state) {
        case state_1: {
            switch (c) {
                case 'a' : {
                    c := nextChar();
                    state := state_2;
                }
            }
        }
        case state_2: {
            switch (c) {
                case 'b' : {
                    c := nextChar();
                    state := state_2;
                }
                default : {
                    return; /* accept */
                }
            }
    }
}
```




... and see th
character ...
Brackets mean we
don't consume any input
next state
state
char1
char2
other
accept
1
2
2
2
2
[3]
3
√
If you're
in this
state ...
... then continue
in this state
A tick here means this
is an accepting state

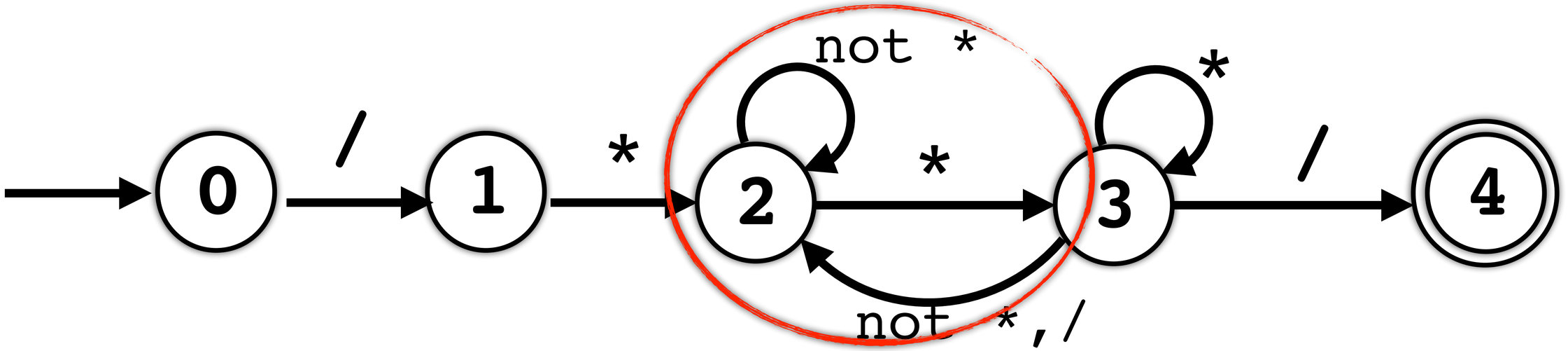
```
state := 1
c := first char
while (not ACCEPT[state]) {
    newstate := NEXTSTATE[state,c]
    if ADVANCE[state,c] {
        c := nextChar()
    }
    state := newstate
}
```

Set the start state and get the first input character

Are we done?

Should we move to the next character?

Move to the next state



state	/	*	other	accept
0	1			
1		2		
2	2	3	2	
3	4	3	2	
4				√

```
static final int SLASH = 0;
static final int STAR  = 1;
static final int OTHER = 2;
static final int EOF   = 3;
```

```
static boolean[] ACCEPT =
  {false,false,false,
   false,true};
```

```
static boolean[][] ADVANCE={
//   "/"    "*"     other
    {true, true, true},
      ...    ...    ...
};
```

state	/	*	other	accept
0	1			
1		2		
2	2	3	2	
3	4	3	2	
4				√

```
static int[][] NEXTSTATE={
// "/"   "*"   other
    { 1,  -1,   -1}, // state 0
    {-1,   2,   -1}, // state 1
    { 2,   3,    2}, // state 2
    { 4,   3,    2}, // state 3
    {-1,  -1,   -1}  // state 4
};
```

```
static String input;
static int current = -1;

static int nextChar() {
    int ch;
    current++;
    if (current >= input.length()) return END;
    switch (input.charAt(current)) {
       case '/' : { ch = SLASH; break;}
       case '*' : { ch = STAR; break;}
       default  : { ch = OTHER; break;}
     }
     return ch;
}
```

```
public static boolean interpret () {
    int state = 0;
    int c = nextChar();
    while ((c != EOF) &&
           (state>=0) &&
            !ACCEPT[state]) {
        int newstate = NEXTSTATE[state][c];
        if (ADVANCE[state][c])
            c = nextChar();
        state = newstate;
    }
    return (state>=0) && ACCEPT[state];
}
```

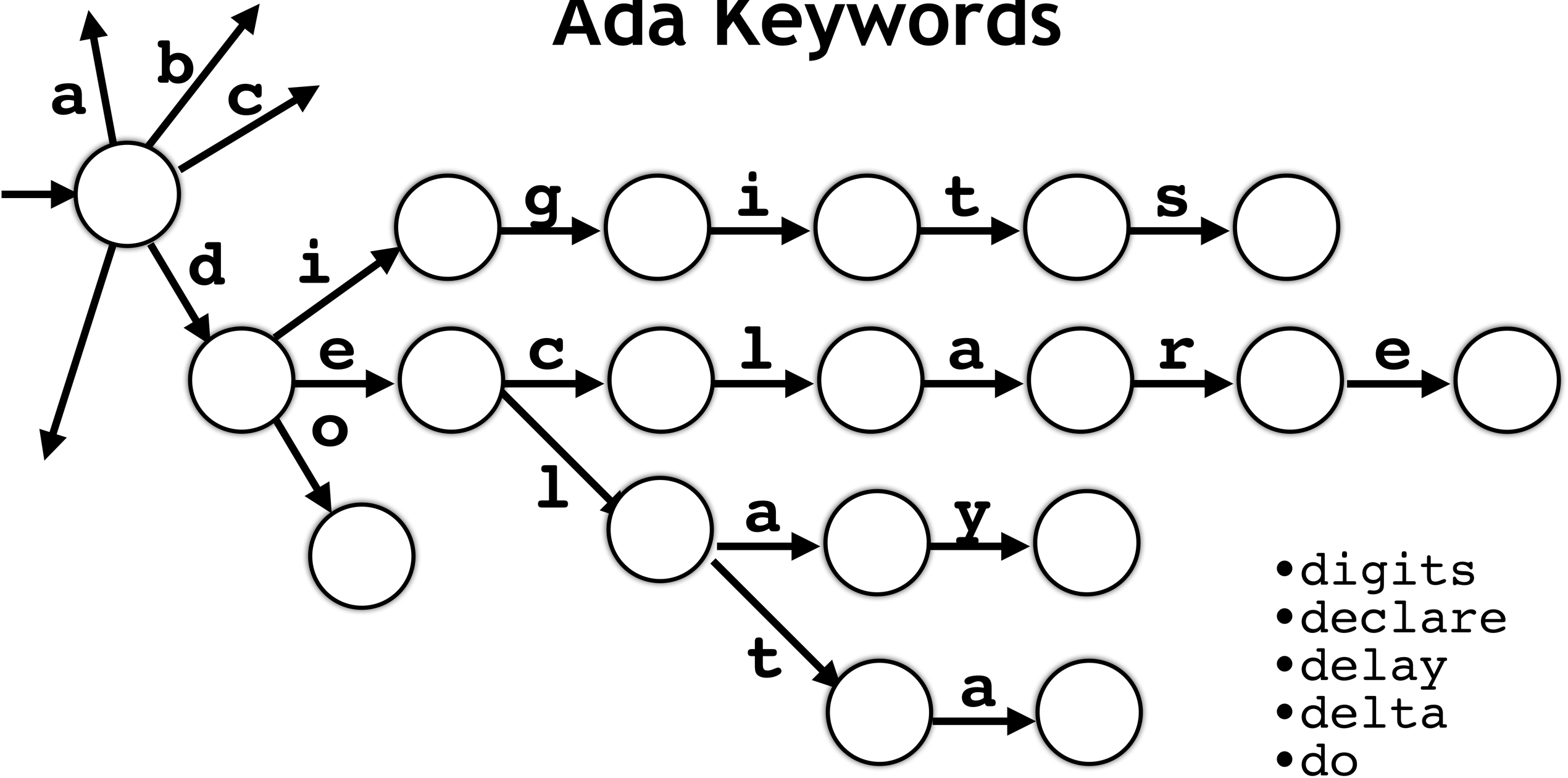
Iterate until the end of input, or we're in an error or accepting state

```
switch (ch) {
switch (ch) {
    case SLASH : ch=nextChar(); state=4; break;
    case STAR  : ch=nextChar(); state=3; break;
    case OTHER : ch=nextChar(); state=2; break;
    default    : return false;
}
            case 1 : { }
            case 2 : { }
            case 3 : { }
            case 4 : return (ch==END);
        }
    }
}
```

	/	*	other	acc
0	1			
1		2		
2	2	3	2	
3	4	3	2	
4				√

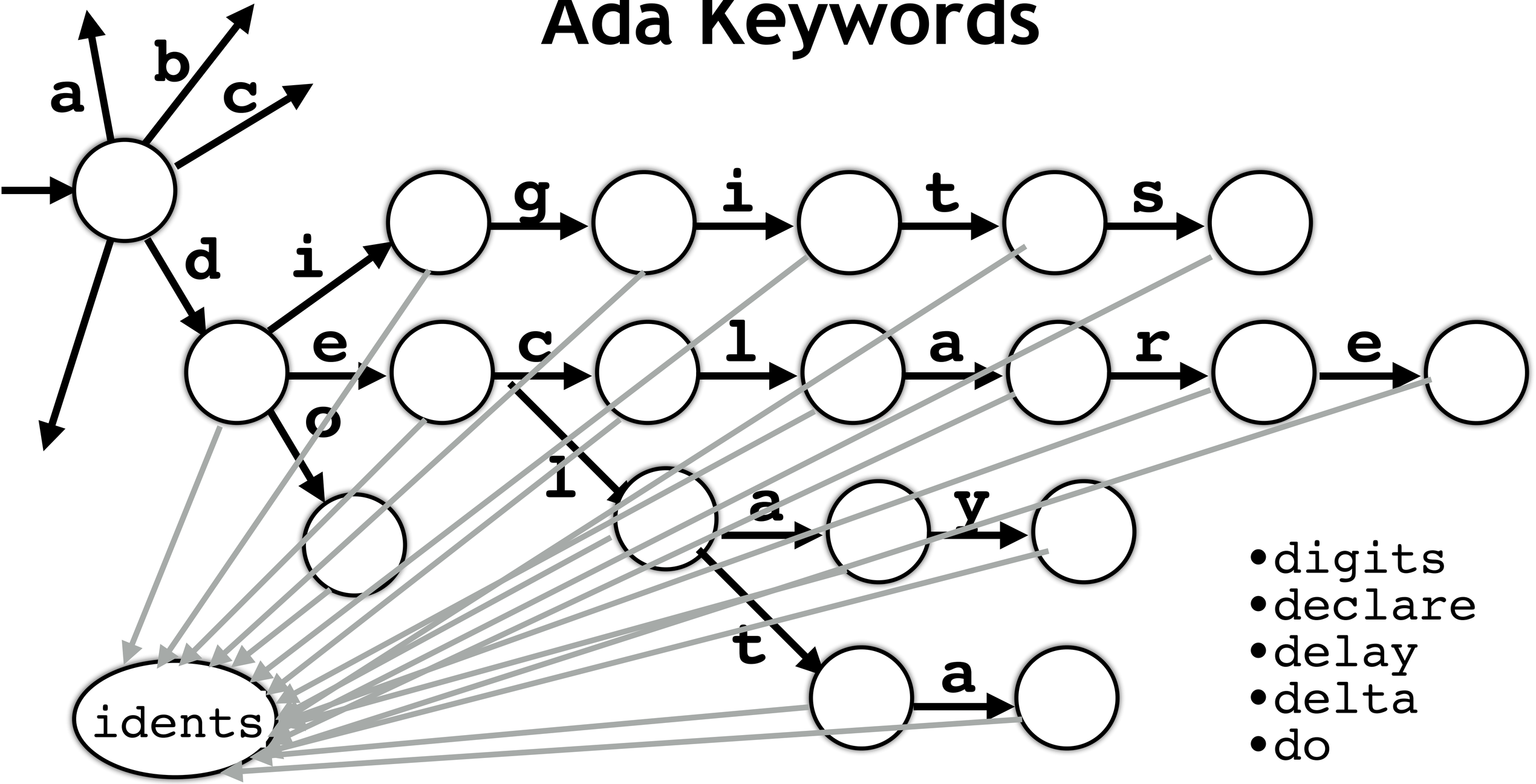
Dealing with Keywords

Ada Keywords

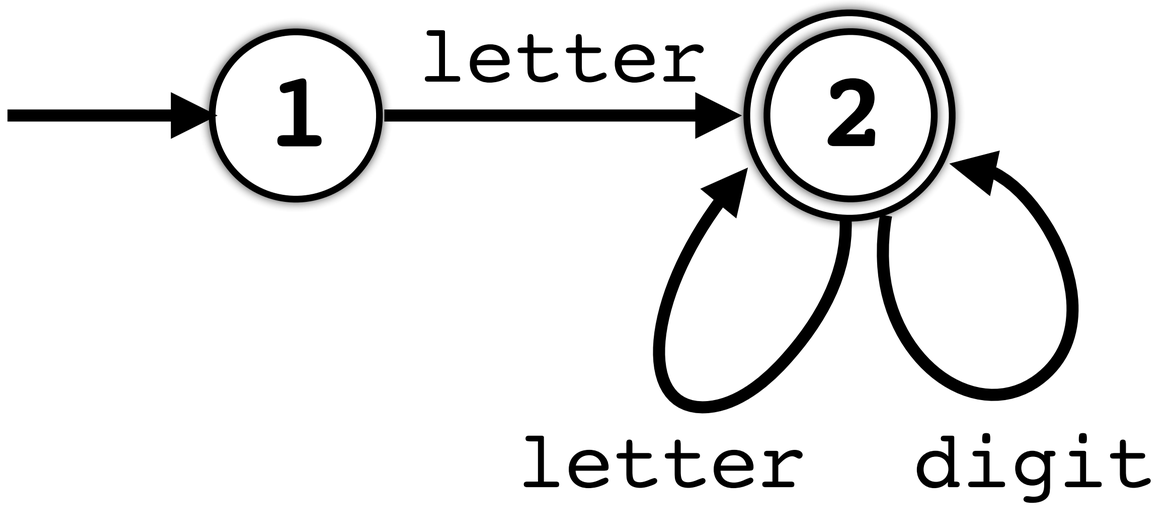


- digits
- declare
- delay
- delta
- do

Ada Keywords



- digits
- declare
- delay
- delta
- do




```
if (is_keyword(s))
    return keyword(s)
else
    return ident(s)
```

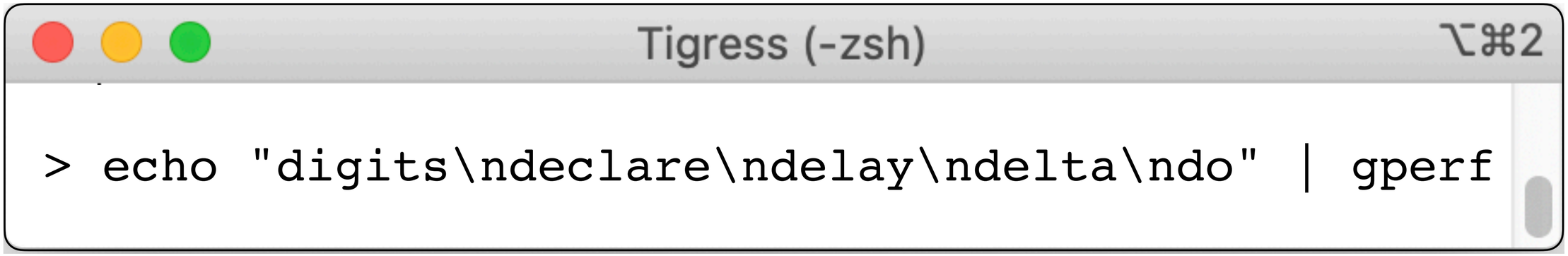
```
bool is_keyword(s) {
  return list_member(s,
    {"digits","declare","delay","delta","do",...}
}
```

Perfect Hashing with `gperf`

gperf takes a list of keywords as input and returns a perfect hash-table (and related search routines) as output.

From the gperf manual (`https://www.gnu.org/software/gperf`):

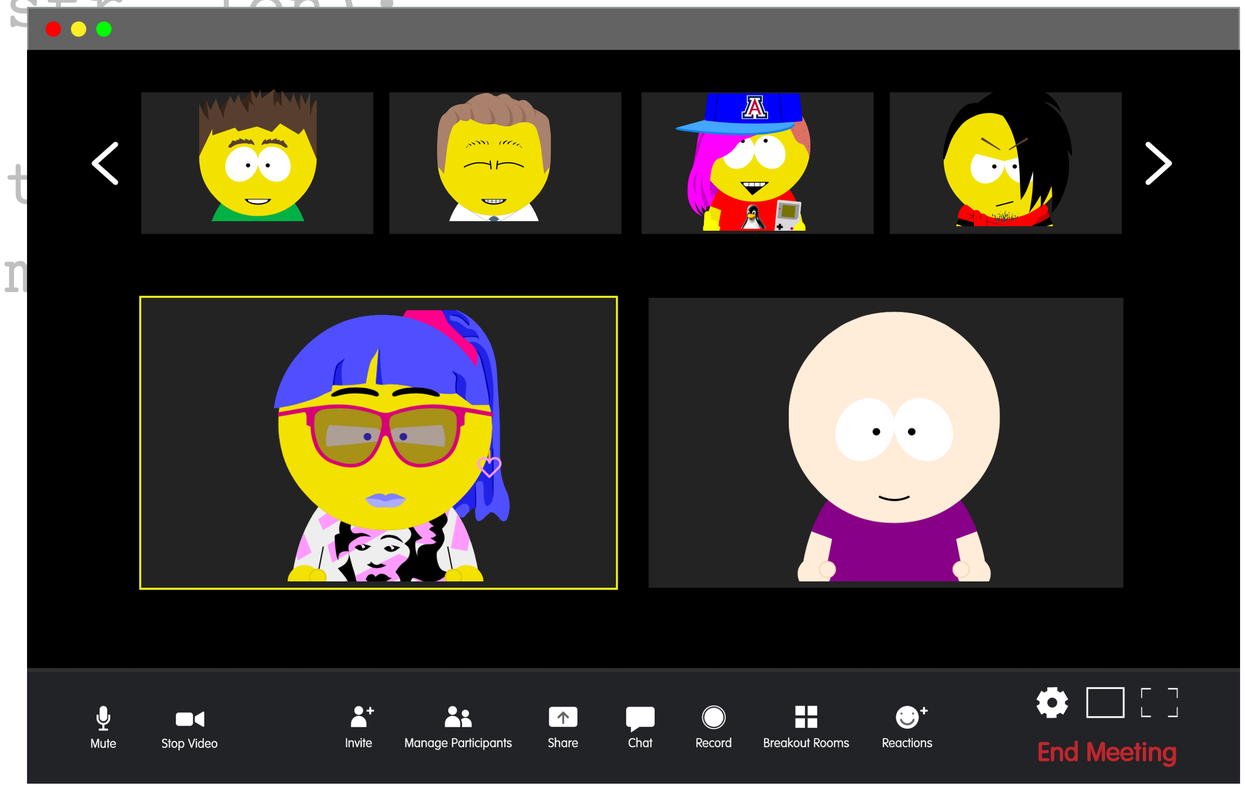
> *The perfect hash function generator gperf reads a set of "keywords" from a keyfile. It attempts to derive a perfect hashing function that recognizes a member of the static keyword set with at most a single probe into the lookup table. If gperf succeeds in generating such a function it produces a pair of C source code routines that perform hashing and table lookup recognition.*

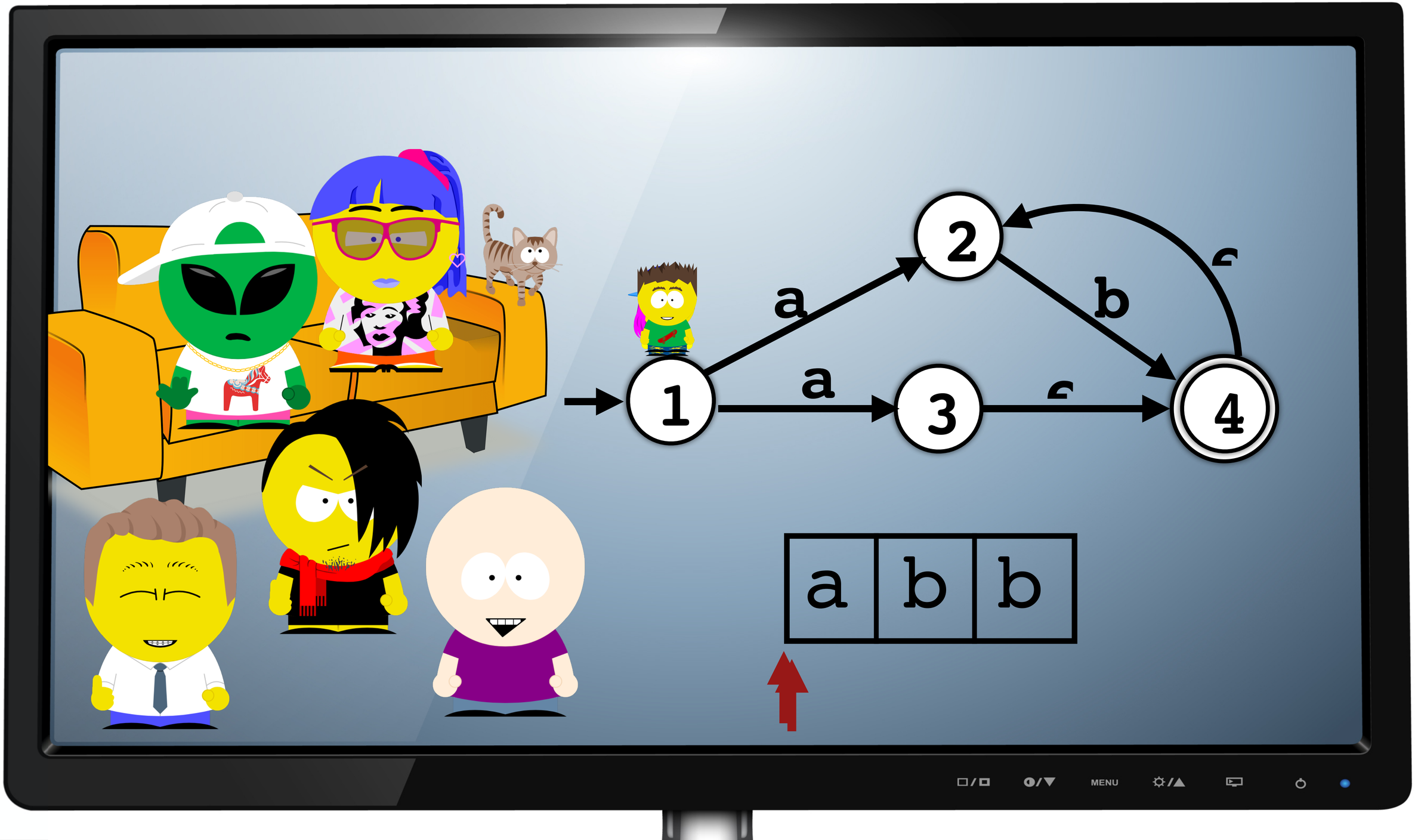



```
static unsigned int hash (const char *str, size_t len) {
  static unsigned char asso_values[] = { ...
      11, 11, 11, 11, 11, 0, 11, 11, 0, 11,
      ... };
  unsigned int hval = len;
  switch (hval) {
      default:  hval += asso_values[(unsigned char)str[3]];
      case 3: case 2:  break;
  }
  return hval;
}
```

```
const char * is_keyword (const char *str, size_t len) {
  static const char * wordlist[] =
    { "", "","do", "", "","delta","digits",
      "declare","", "","delay"};
  if (len <= 7 && len >= 2) {
    unsigned int key = hash (str, len);
    if (key <= 10)  {
      const char *s = wordlist[key];
      if (*str == *s && !strcmp (str + 1, s + 1))
         return s;
      }
  }
  return  0;
}
```

```
const char * in_word_set (const char *str, size_t len) {
  static const char * wordlist[] =
    { "", "","do", "", "","delta","digits",
      "declare","", "","delay"};
  if (len <= 7 && len >= 2) {
    unsigned int key = hash (str, len);
    if (key <= 10)  {
      const char *s = wordlist
      if (*str == *s && !strcm
        return s;
      }
  }
  return  0;
}
```




2
a
b
ε
1
a
3
ε
4
a b b
MENU