CS 315 – Programming Languages
Scanning

Scanning

- Dramatically reduces the number of items to be inspected by the parser
- removes comments, ignores whitespace
- saves text of interesting tokens, e.g., identifiers, literals
- tags tokens with line and column numbers

Let us do a complete example:

ASSIGN: “:=”
PLUS: '+'
MINUS: '-'
TIMES: '*'
DIV: '/'
LPARAN: '('
RPARAN: ')'

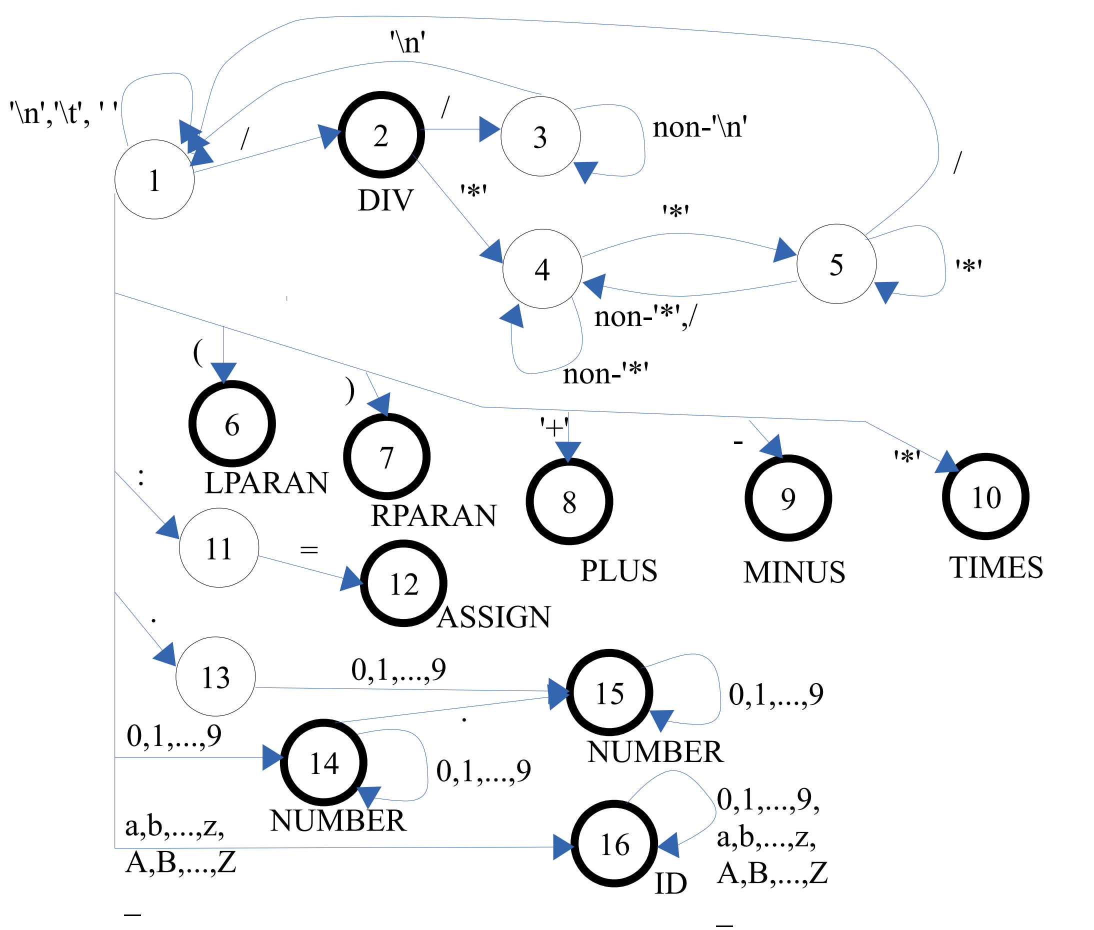
DIGIT: [0-9]
LETTER: [a-zA-Z_]
NEWLINE: '\n'
NONSTAR: [^*]
NONSTARORDIV: [^*/]
NONNEWLINE: [^\n]
WHITESPACE: [\n\t]

ID: LETTER (LETTER | DIGIT) *
NUMBER: DIGIT + | DIGIT* ('.' DIGIT | DIGIT '.') DIGIT*
COMMENT: “//” NONNEWLINE* NEWLINE
| “/*” (NONSTAR | '*'+ NONSTARORDIV)* '*'+ '/'

How do we use these regular expressions to divide our input into tokens?

- We create a state machine out of it
- Final states (bold) correspond to the tokens (underlined ones above)
- Transitions correspond to the consumed input characters

For now, we will create a state diagram manually, using our intuition.



Longest possible match is taken as the next token.

Let's do a few examples:

(xy +10.2) / z// abc
x :=3$

s1 '(' → **s6** ' ' <stop>
=> LPARAN [**s6**] (
s1 ' ' → s1 'x' → **s16** 'y' → **s16** ' ' <stop>
=> ID [**s16**] xy
s1 ' ' → s1 '+' → **s8** '1 ' <stop>
=> PLUS [**s8**] +
s1 '1' → **s14** '0' → **s14** '.' → **s15** '2' → **s15** ')' <stop>
=> NUMBER [**s15**] 10.2
s1 ')' → **s7** ' ' <stop>
=> RPARAN [**s7**])
s1 ' ' → s1 '/' → **s2** ' ' <stop>
=> DIV [**s2**] /
s1 ' ' → s1 'z' → **s16** '/' <stop>
=> ID [**s16**] z
s1 '/' → **s2** '/' → s3 ' ' → s3 'a' → s3 'b' → s3 'c' → s3 '\n' → s1 'x' → **s16** ' ' <stop>
=> ID [**s16**] x
s1 ' ' → s1 ':' → s11 '=' → **s12** '3 ' <stop>
=> ASSIGN [**s12**] :=
s1 '3' → **s14** '$ ' <stop>
=> NUMBER [**s14**] 3

/*a$

s1 '/' → **s2** '*' → s4 'a' → s4 $ <stop>
=> DIV [**s2**] /
s1 '*' → **s10** 'a' <stop>
=> TIMES *
s1 'a' → **s16** '$' <stop>
=> ID [**s16**] a

The state machine we have constructed is known as a Deterministic Finite Automata (DFA). It has in important property: For each state, given an input character, there is either nowhere to go, or only a single next state to move into. In that sense, it is deterministic. With such a DFA, it is easy to construct a scanner. It could be hand-crafted, where for each state, we write a function. Or it could be an automatically created scanner that uses a table-driven implementation. Such scanners are created by a scanner generator, such as lex (will be covered later).

An important question remains: How do we create the DFA, given the regular expressions? This is what we will cover next.

Creating a Finite Automata

Our goal is: Regular Expression (RE) → Deterministic Finite Automata (DFA)

There are two kinds of Finite Automata (FA):

- DFA, deterministic
    - no ambiguity about the next step
- NFA, non-deterministic
    - more than one transition with the same character are allowed
    - epsilon transitions are allowed
        - these are transitions that do not consume an input character

We prefer a DFA, as it is more easily converted into code, and is also more efficient to execute. We will carry out the transformation from REs into DFAs, as follows:

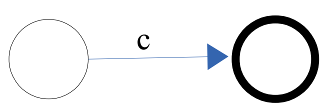
RE → NFA → DFA → optimized DFA (reduced size)

Why the extra step that converts into NFA?

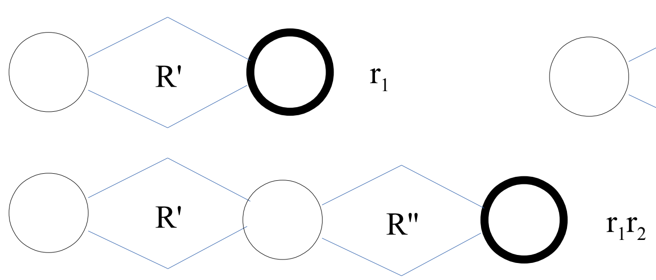
- Because the transformation from RE into an NFA is very straightforward and easily expressed systematically
- There is a known algorithm to convert NFAs into DFAs

RE → NFA

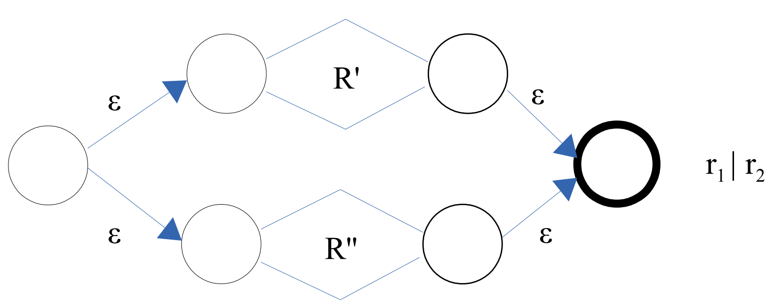
a) *Base case*: regular expression that consists of a single character 'c'



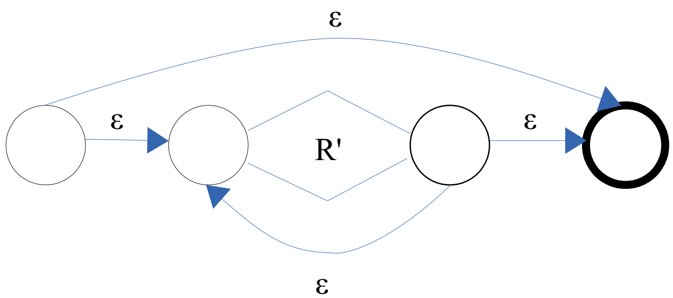
b) *Concatenation* of two regular expressions r_1 and r_2, forming the regular expression r_1r_2



c) *Alternation* of two regular expressions r_1 and r_2, forming the regular expression $r_1 | r_2$

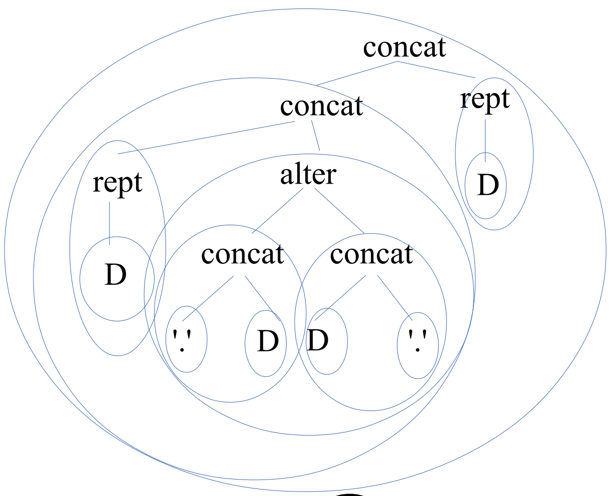


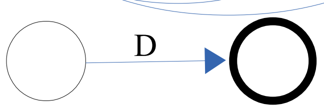
d) Kleene Closure



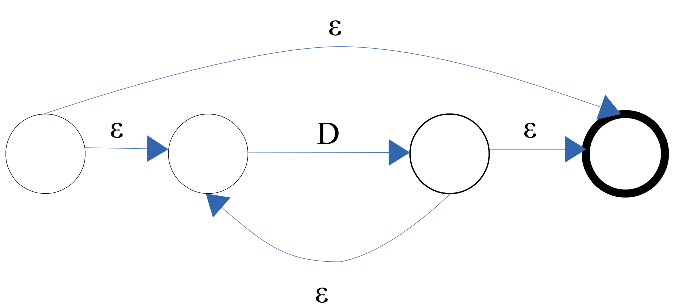
r_1 *

Example: D*('.'D | D '.') D*

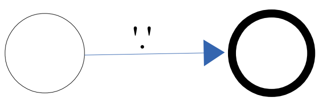


1) 

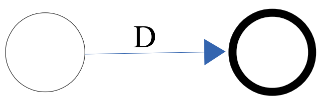
D

2) 

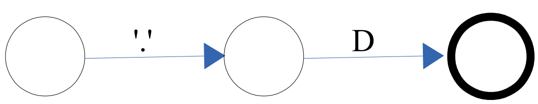
D* as (1)*

3) 

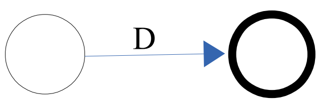
'.'

4) 

D

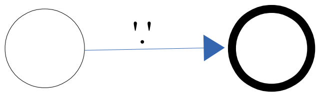
5) 

'.' D as (3) (4)

6) 

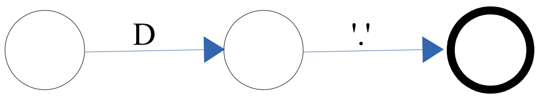
D

7)



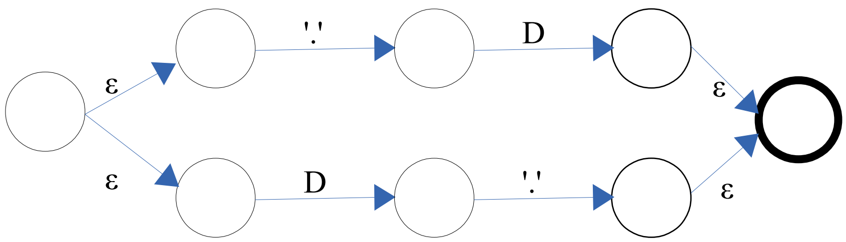
'.'

8)



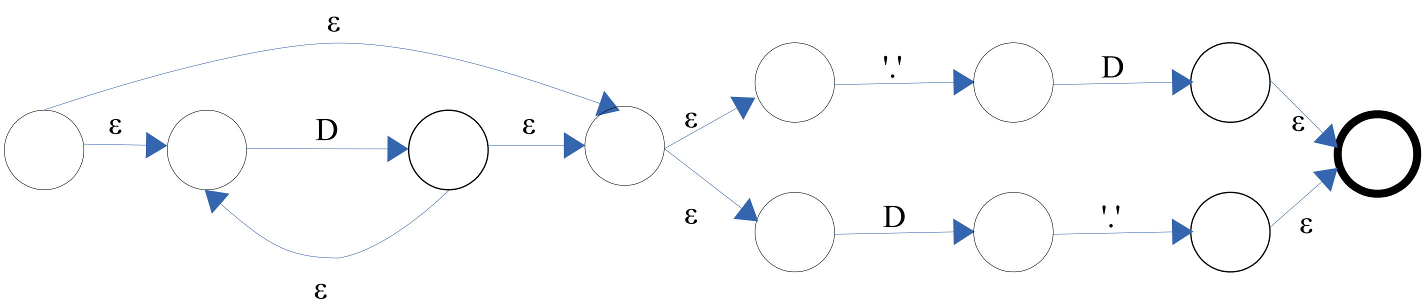
D '.'   as (6) (7)

9)

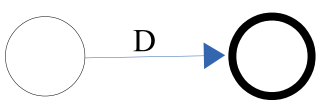


('.' D | D '.')   as (5) | (8)

10) D*('.'D | D '.')   as (2) (9)

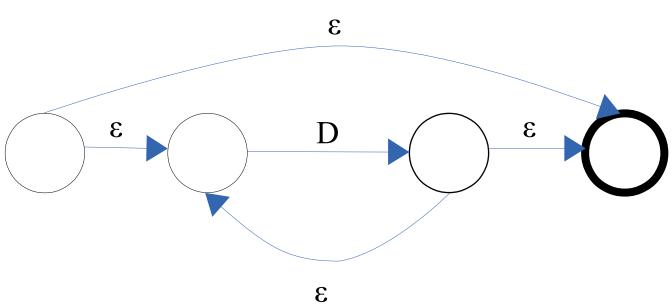


11)



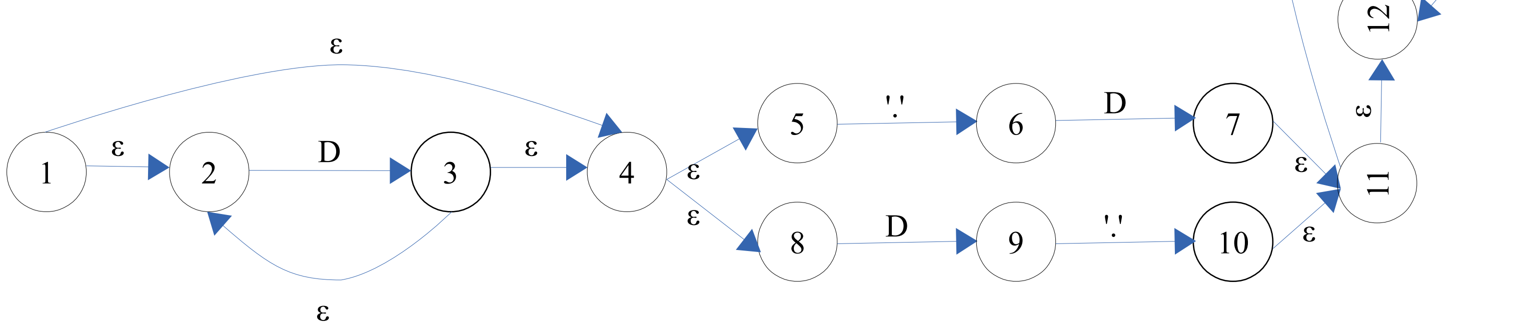
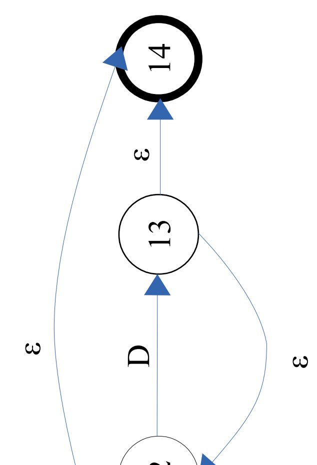
D

12)



D *   as (11) *

13) D*('.'D | D '.') D*   as (10) (12)

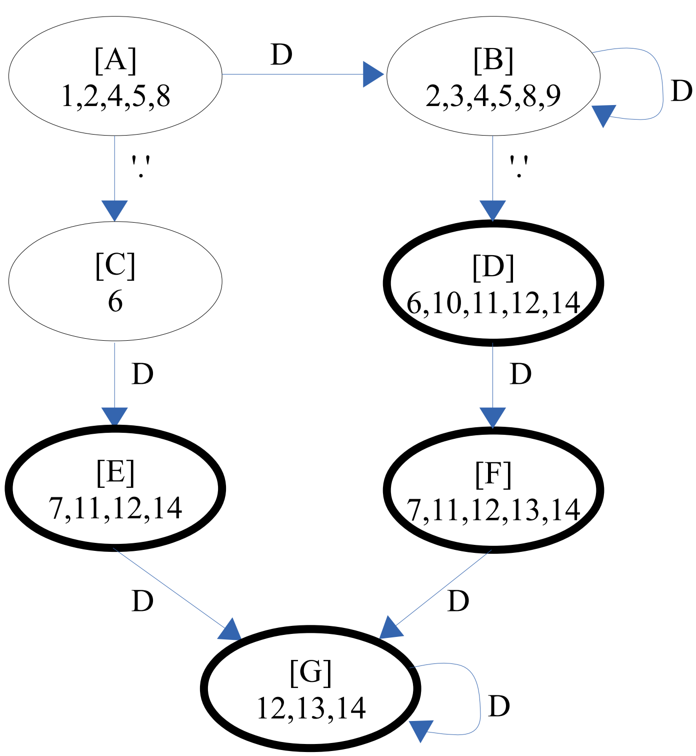
Converting an NFA into a DFA

Definitions:

- *epsilon-closure* of a state: subset of states you can reach from this state using zero or more epsilon transitions.
- *c-transition-set* of a state: subset of states you can reach from this state by following a singe c-transition.

Algorithm:

1. Find the epsilon-closure of the starting state of the NFA. Call the resulting subset of NFA states I.
2. Add I as a state to the DFA and make it the starting state.
3. While there are unprocessed states in the DFA, pick one such state, say S.
    1. For each character c:
        1. Find the c-transition-set of S, by taking the union of the c-transition-sets of the individual NFA states in S. Call the resulting subset of NFA states as S^-.
        2. Find the epsilon-closure of S^-, by taking the union of the epsilon-closures of the individual NFA states in S^-. Call the resulting subset of NFA states S^+.
        3. Make S^+ a state of the DFA, unless it already exists.
        4. Add $S \rightarrow S^+$ as a c-transition to the DFA
    2. Mark S as processed
4. Mark all DFA states that contain a final state of the NFA as a final state of the DFA.



Note: In class, we show this up to 2 or three states, as follows:

ε-closure of 1:
1,2,4,5,8

We add a starting state [A]: 1,2,4,5,8

The only state, which is unprocessed, is [A], so

For state [A]
For character D
D-transition of [A] is the union of
for 1: {}
for 2: {3}
for 4: {}
for 5: {}
for 8: {9}
which is {3, 9}
ε-closure of {3,9} is the union of
for 3: {2, 3, 4, 5, 8}
for 9: {9}
which is {2,3,4,5,8,9}
Add a new state [B] (2,3,4,5,8,9)
Add a D-transition from [A] to [B]
For character '.'
'.'-transition of [A] is the union of
for 1: {}
for 2: {}
for 4: {}
for 5: {6}
for 8: {}
which is {6}
ε-closure of {6} is the union of
for 6: {6}
which is {6}
Add a new state [C] (6)
Add a '.'-transition from [A] to [C]

For state [B]
For character D
D-transition of [B] is the union of
for 2: {3}
for 3: {}
for 4: {}
for 5: {}
for 8: {9}
for 9: {}
which is {3, 9}
ε-closure of {3,9} is the union of
for 3: {2, 3, 4, 5, 8}

for 9: {9}
which is {2,3,4,5,8,9}
Add a new state [B] (2,3,4,5,8,9)
Add a D-transition from [B] to [B]
For character '.'
'.'-transition of [B] is the union of
for 2: {}
for 3: {}
for 4: {}
for 5: {6}
for 8: {}
for 9: {10}
which is {6,10}
ε-closure of {6,10} is the union of
for 6: {6}
for 10: {10,11,12,14}
which is {6,10,11,12,14}
Add a new state [D] (6,10,11,12,14)
Add a '.'-transition from [B] to [D]
Make [D] a final state as it contains 14

Minimizing a DFA (not shown in class)

- Initially divide the states into two equivalence classes: final and non-final
- For each character c:
  - For each equivalent class A
    - If there are states in the equivalence class A that have transitions to different equivalence classes, divide the equivalence class A into multiple equivalence classes A_1, A_2, …, A_n so that all states in A_k to to the same equivalence classes on c
- Continue until no changes