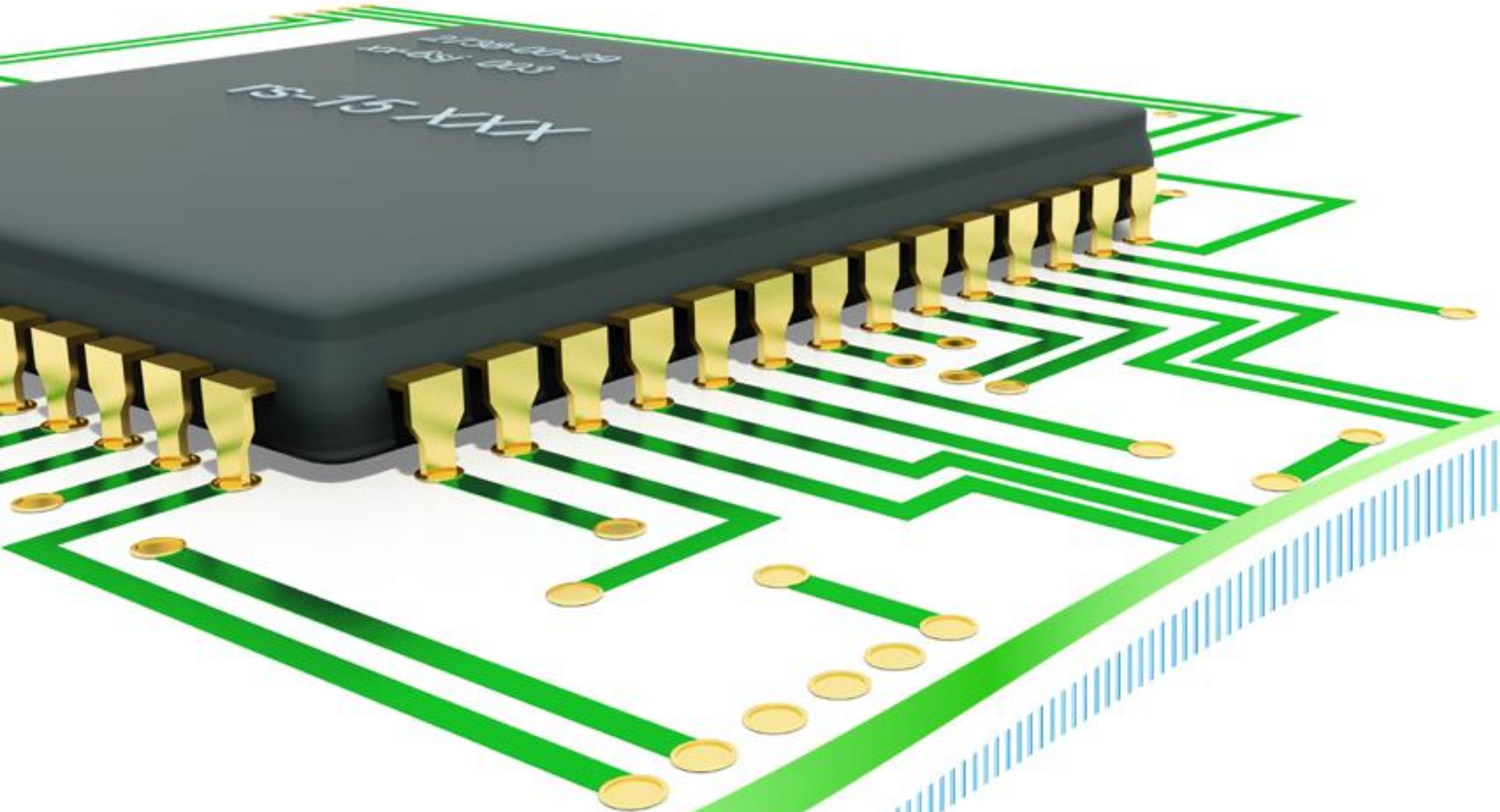
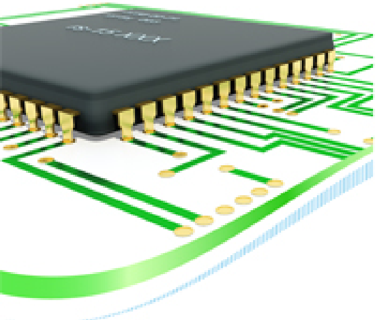


Lexical Analysis



Definition

Lexical analysis is the process of identifying the tokens which are basic building blocks of a given language.

Usual Scheme

- **No Context**
- **One Pass Analysis**
- **Buffer Management**

- **Buffer Size I/O Trade Off**

- **Unusual Schemes?**

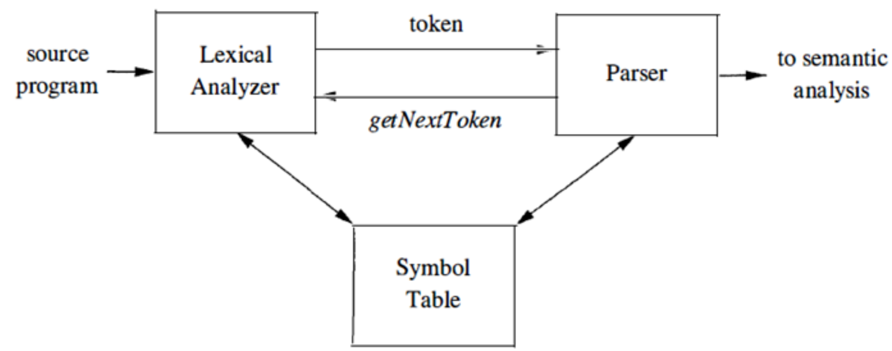
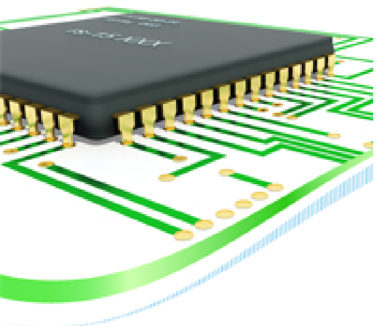
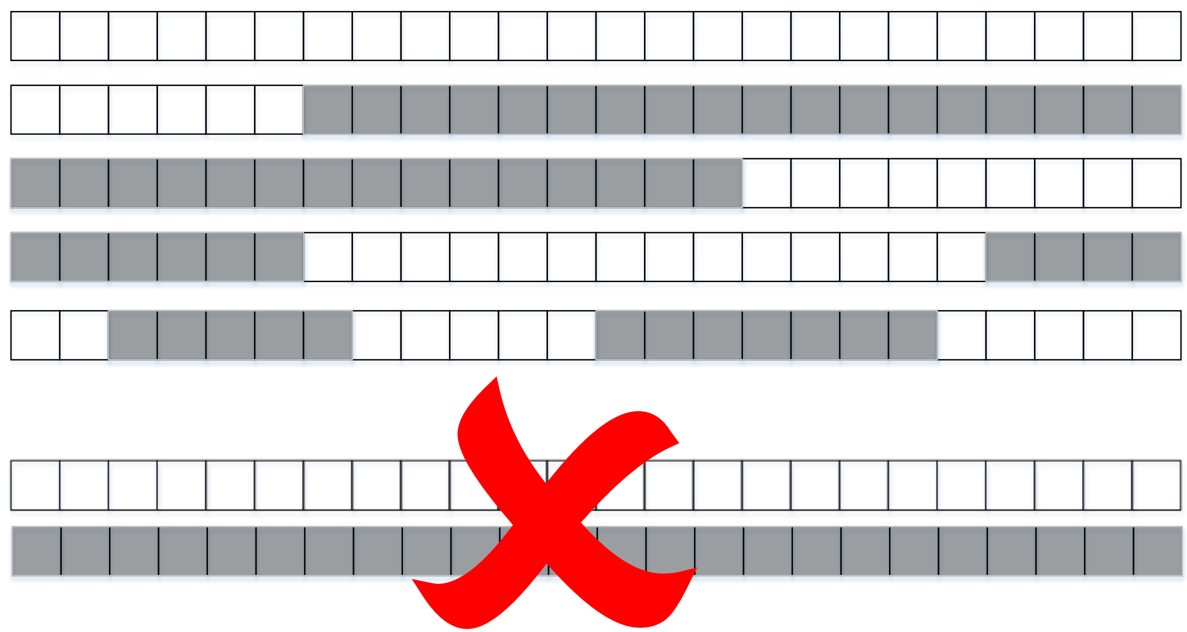


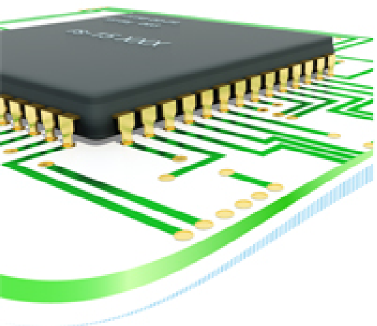
Diagram from “Aho, A.V, Ullman J.D, Sethi R., Lam M.S; Dragon Book”

Scanning
Strings



- String
- Prefix
- Suffix
- Substring
- Subsequence
- Proper Versions





Scanning

Patterns, Lexemes, Tokens

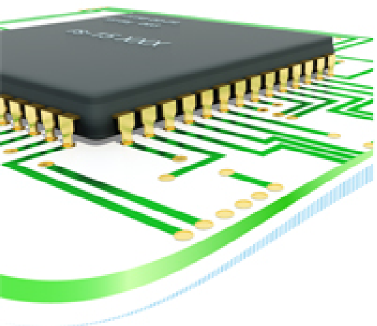
Sample Patterns Informally Described

- An id is a string of characters starting with starters character may continue with string of id-continuation characters. Valid starter character must be in set a..z, A..Z, or underscore. Id-continuation character must be in set a..z, A..Z, 0..9 or underscore.

- A simple number starts with nonzero decimal digit which may be followed by zero or more decimal digits.

Sample Patterns Formally Described

- `Id: [A-Za-z_][A-Za-z_0-9]*`
- `SimpleNumber: [1-9][0-9]*`



Scanning

Patterns, Lexemes, Tokens

Sample Input and Lexemes Identified

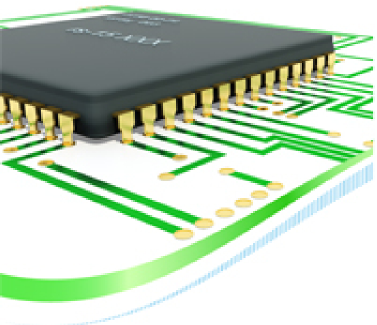
```
memberNo_1 = 1340 ;
```

Sample Tokens Allocated and Streamed

```
Id -------- SimpleNumber --------
```

```
Token Type Id: Simple Number
Marker / copy: 1340
Position in source: …
```

```
Token Type Id: Id
Marker / copy: memberNo_1
Position in source: …
```



Scanning

Related Side Processing

- Comment Processing
  - With Lexical Analyzer
  - With Preprocessor
  - Other

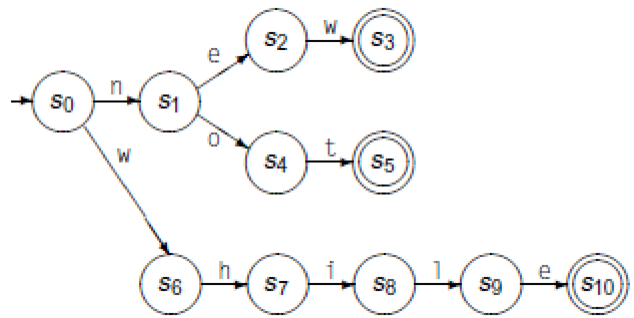
- Preprocessors
  - As First Instance Scanners
  - Preprocessor / Lexer Source Flow

- Documentation Processors
  - Comments as Language Containers

Regular Expressions

Definition, Applicability

- **Regular Languages, Finite State Automata.**

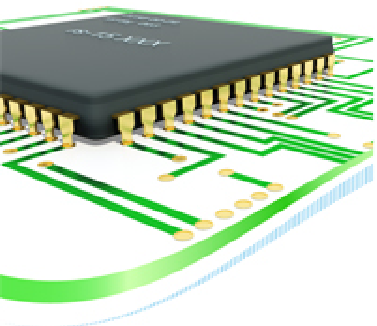


- **Ease of Specification and Maintenance.**

```
kw: "while" | "n" ("ew"|"ot")
```

- **Ad-hoc Recognition or Automatic Generation of Efficient Recognizers.**

Diagram from “Cooper, K.D., Torczon, L.; Engineering A Compiler”



Regular Expressions

Rule Based Definition

- **Epsilon**

  R: ε

- **Symbol / Set**

  R: α, $\alpha \in A$

  R: $\{\alpha : \alpha \in A\}$

- **Concatenation**

  R: R_1R_2

- **Alternation**

  R: $R_1 \mid R_2$

- **Kleene Closure**

  R: R1*

- **Precedence control**

  R: (R_1)

- **Practical notations**

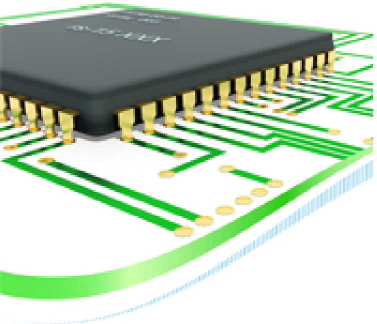
  R: R_1+

  R: $R_1?$

- **Operations out of notations**

  Intersection

  Negation



Regular Expressions

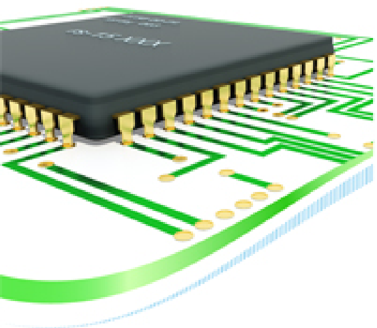
- **Tools (Lex, Flex, Antlr, …)**
- **Code Generation**
- **Conventions and Toolchains**

6 Patterns

The patterns in the input (see Section 5.2 [Rules Section], page 7) are written using an extended set of regular expressions. These are:

`'x'`	match the character 'x'
`'.'`	any character (byte) except newline
`'[xyz]'`	a *character class*; in this case, the pattern matches either an 'x', a 'y', or a 'z'
`'[abj-oZ]'`	a "character class" with a range in it; matches an 'a', a 'b', any letter from 'j' through 'o', or a 'Z'
`'[^A-Z]'`	a "negated character class", i.e., any character but those in the class. In this case, any character EXCEPT an uppercase letter.
`'[^A-Z\n]'`	any character EXCEPT an uppercase letter or a newline
`'[a-z]{-}[aeiou]'`	the lowercase consonants
`'r*'`	zero or more r's, where r is any regular expression
`'r+'`	one or more r's
`'r?'`	zero or one r's (that is, "an optional r")
`'r{2,5}'`	anywhere from two to five r's
`'r{2,}'`	two or more r's
`'r{4}'`	exactly 4 r's

Excerpt from flex manual. https://epaperpress.com/lexandyacc/download/flex.pdf



Look Ahead!

- **C++ template syntax:**

```
Foo<Bar>
```

- **C++ stream syntax:**

```
cin >> var;
```

- **But there is a conflict with nested templates:**

```
Foo<Bar<Bazz>>
```

Closing templates, not stream

Excerpt from https://web.stanford.edu/class/cs143/lectures/lecture03.pdf

Look Ahead!

- **IO / Buffering Techniques**

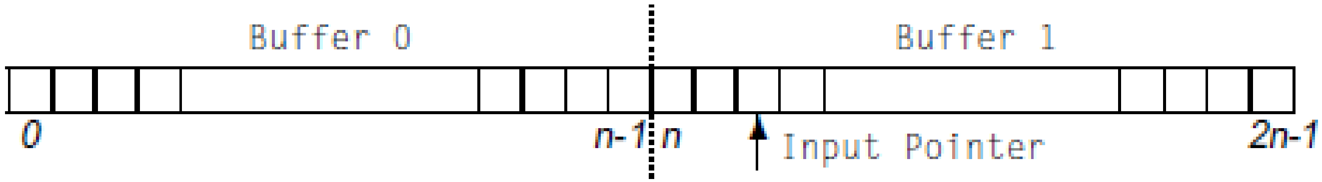


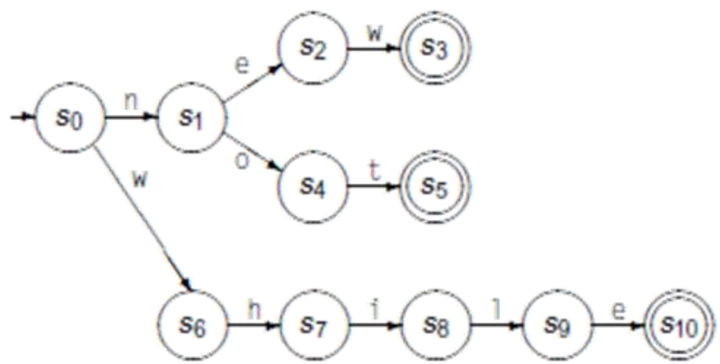
Diagram from “Cooper, K.D., Torczon, L.; Engineering A Compiler”

FSA Based Recognition

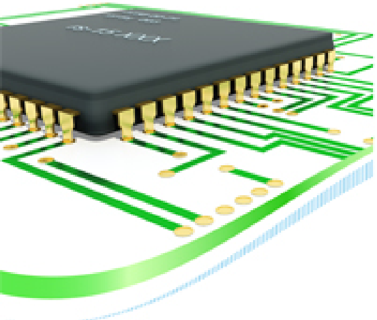
A Reminder on Finite State Automata

$(S, \Sigma, \delta, s_0, S_A)$

S: Set of States
Σ: Alphabet
δ: Transition Function
s_0 : Start State
S_A : Set of Accepting States



$$\delta: S \times \Sigma \rightarrow S$$



FSA Based Recognition

Nondeterministic Finite Automata - NFA

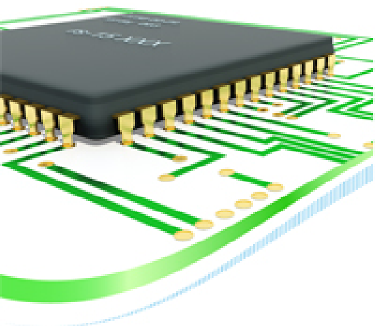
$(S, \Sigma, \delta, s_0, S_A)$

S: Set of States
Σ: Alphabet
δ: Transition Function
s_0 : Start State
S_A : Set of Accepting States

$$\delta: S \times (\Sigma \cup \varepsilon) \rightarrow P(S)$$

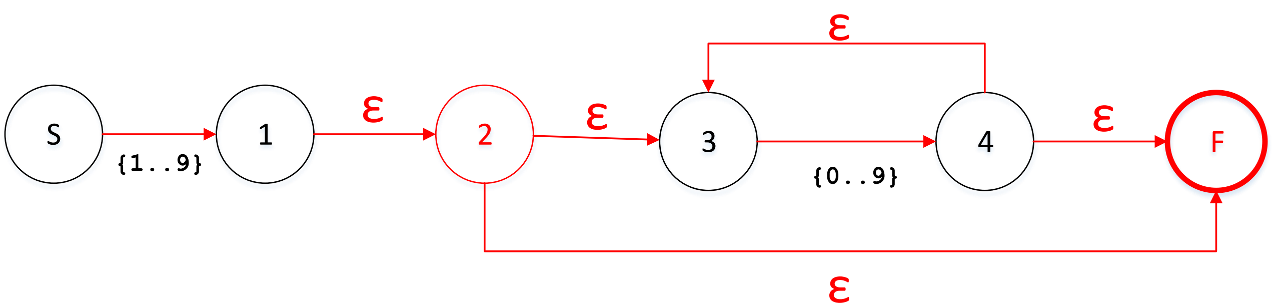
or

$$\delta: S \times (\Sigma \cup \varepsilon) \rightarrow 2^S$$



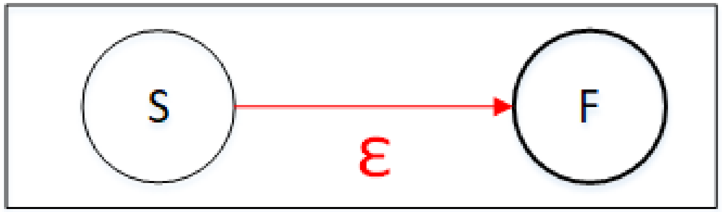
Thompson's Construction

SimpleNumber: [1-9][0-9]*



Epsilon

R: ε

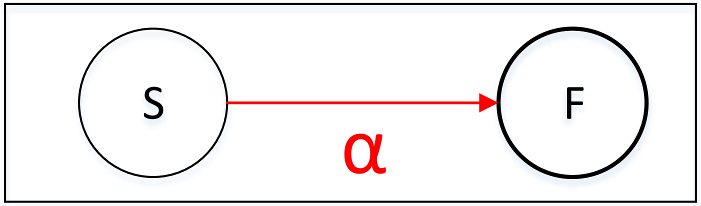


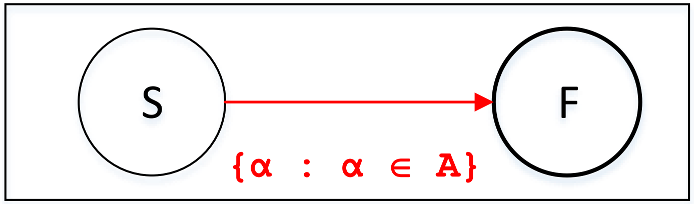
Thompson's Construction

Symbol / Set

R: α, α ∈ A

R: {α : α ∈ A}

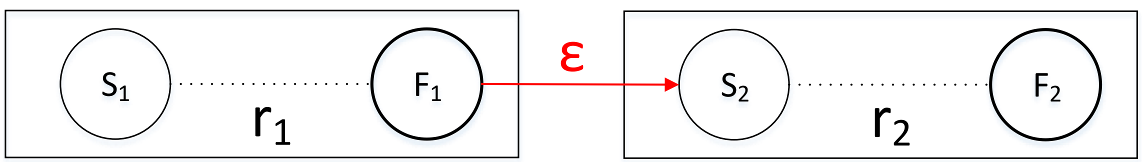




Thompson's Construction

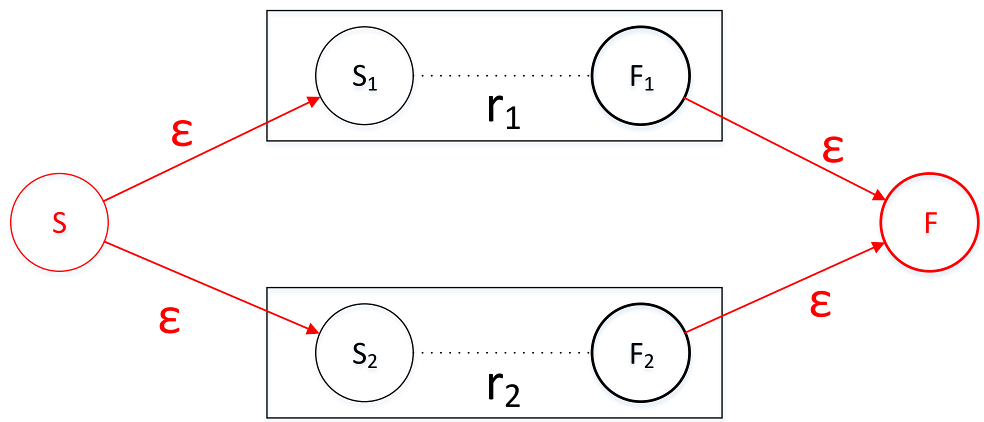
Concatenation

R: R_1R_2



Alternation

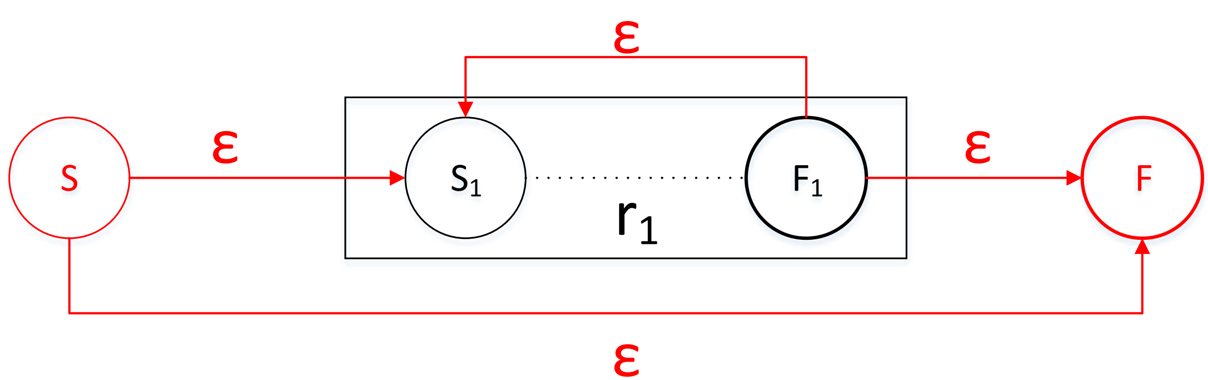
R: R_1 | R_2



Thompson's Construction

Kleene Closure

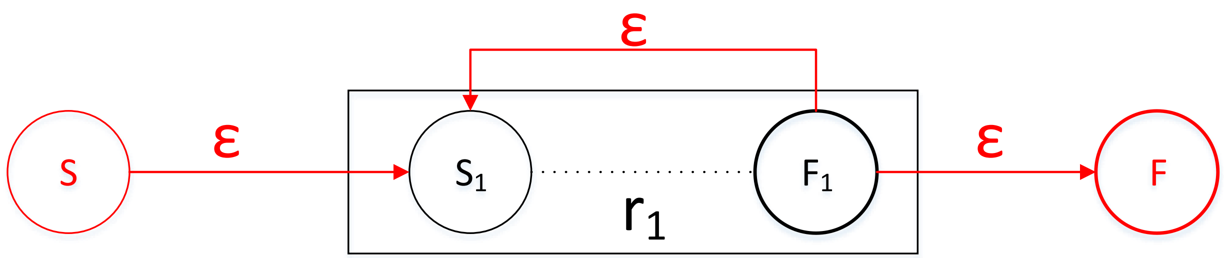
`R: R1*`



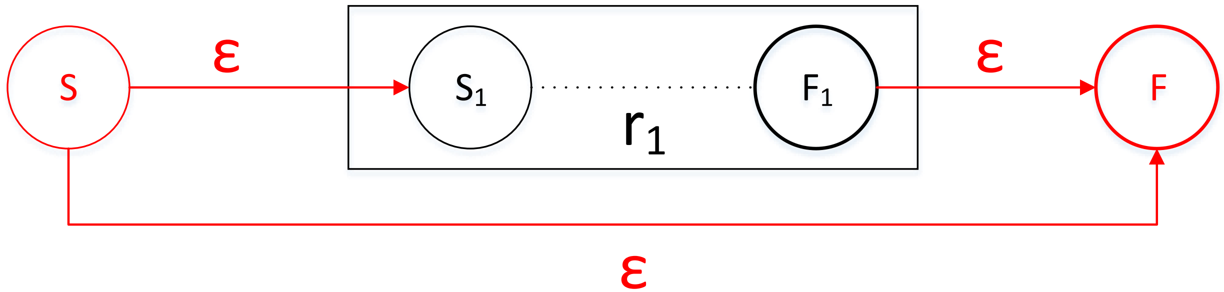
Thompson's Construction

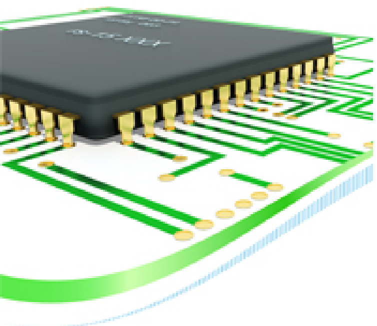
- **Practical notations**

R: R_1+



R: R_1?





NFA

As a recognizer

- **Describe S and F?**
- **What is ε-closure?**
- **What is equivalent of *move* in formal definition?**
- **What is the significance of final check?**

$S = \epsilon\text{-}closure(s_0);$
$c = nextChar();$
while (c != **eof**) {
    $S = \epsilon\text{-}closure(move(S, c));$
    $c = nextChar();$
}
if ($S \cap F$!= $\emptyset$) **return** "yes";
else return "no";

Algorithm from “Aho, A.V, Ullman J.D, Sethi R., Lam M.S; Dragon Book”

DFA

From NFA – Subset Construction

```
initially, ϵ-closure(s0) is the only state in Dstates, and it is unmarked;
while ( there is an unmarked state T in Dstates ) {
        mark T;
        for ( each input symbol a ) {
                U = ϵ-closure(move(T, a));
                if ( U is not in Dstates )
                        add U as an unmarked state to Dstates;
                Dtran[T, a] = U;
        }
}
```

```
q0 ← ϵ-closure({n0});
Q ← q0;
WorkList ← {q0};

while (WorkList≠∅) do
    remove q from WorkList;
    for each character c∈Σ do
        t ← ϵ-closure(Delta(q,c));
        T[q,c] ← t;
        if t ∉ Q then
            add t to Q and to WorkList;
    end;
end;
```

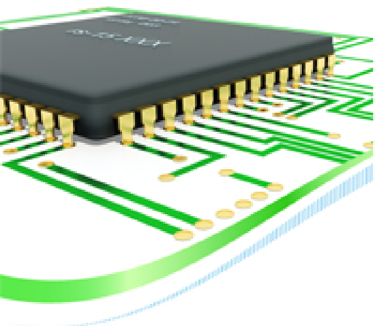
Your thoughts on complexity of construction and complexity?

What should a DFA based recognizer look like?

Algorithms from
“Cooper, K.D., Torczon, L.; Engineering A Compiler” on the left
“Aho, A.V, Ullman J.D, Sethi R., Lam M.S; Dragon Book” on the right

DFA

Construction Without NFA



initialize $Dstates$ to contain only the unmarked state $firstpos(n_0)$,
    where n_0 is the root of syntax tree T for $(r)\#$;
while (there is an unmarked state S in $Dstates$) {
    mark S;
    for (each input symbol a) {
        let U be the union of $followpos(p)$ for all p
            in S that correspond to a;
        if (U is not in $Dstates$)
            add U as an unmarked state to $Dstates$;
        $Dtran[S, a] = U$;
    }
}

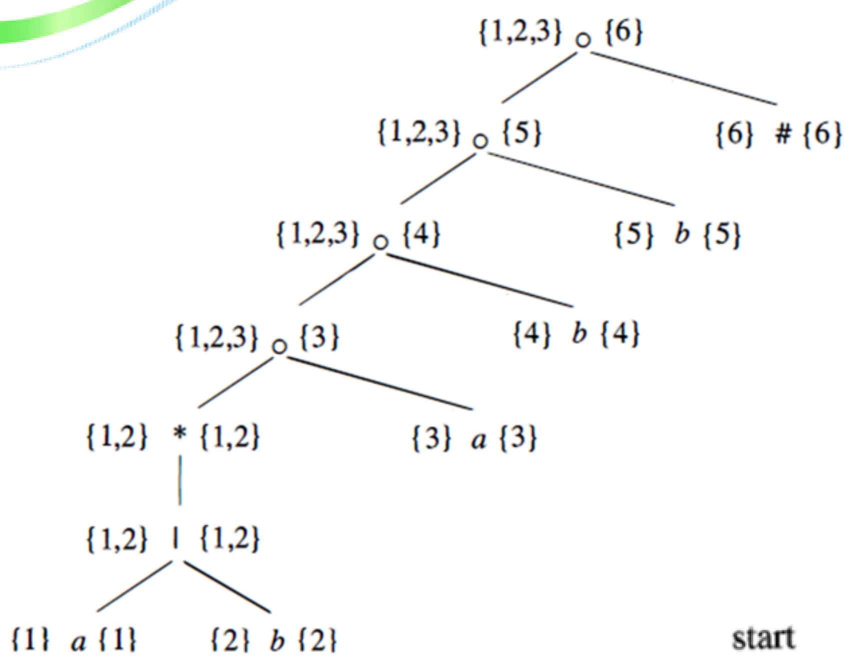
NODE n	$nullable(n)$	$firstpos(n)$
A leaf labeled ϵ	**true**	$\emptyset$
A leaf with position i	**false**	$\{i\}$
An or-node $n = c_1 \vert c_2$	$nullable(c_1)$ **or** $nullable(c_2)$	$firstpos(c_1) \cup firstpos(c_2)$
A cat-node $n = c_1c_2$	$nullable(c_1)$ **and** $nullable(c_2)$	**if** ($nullable(c_1)$) $firstpos(c_1) \cup firstpos(c_2)$ **else** $firstpos(c_1)$
A star-node $n = c_1{}^*$	**true**	$firstpos(c_1)$

Algorithm from
"Aho, A.V, Ullman J.D, Sethi R., Lam M.S;
Dragon Book"

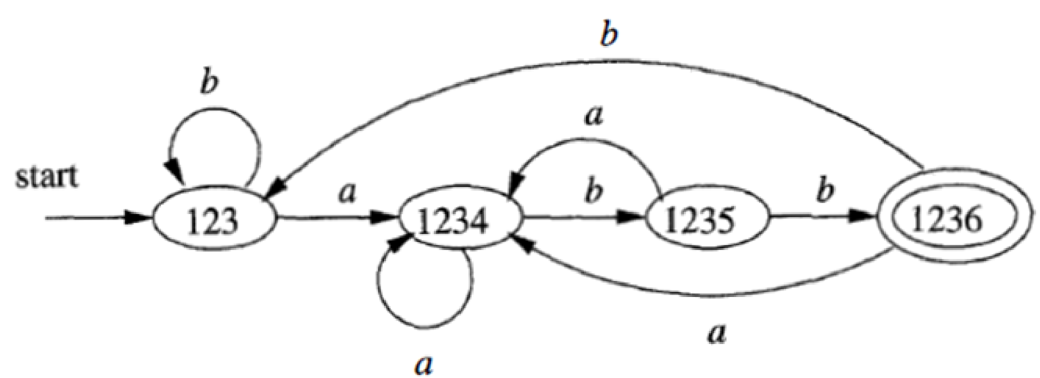
DFA

Construction Without NFA

(a|b)*abb#



NODE n	$followpos(n)$
1	$\{1, 2, 3\}$
2	$\{1, 2, 3\}$
3	$\{4\}$
4	$\{5\}$
5	$\{6\}$
6	$\emptyset$



Excerpts from “Aho, A.V, Ullman J.D, Sethi R., Lam M.S; Dragon Book”

DFA

Minimization

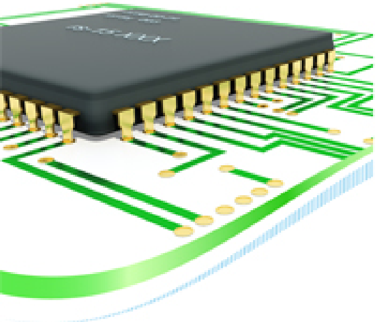
Hopcroft's Algorithm

```
T ← {D_A, {D − D_A} };
P ← Ø

while (P ≠ T) do
    P ← T;
    T ← Ø;

    for each set p ∈ P do
        T ← T ∪ Split(p);
    end;

end;
```

```
Split(S) {
    for each c ∈ Σ do
        if c splits S into s1 and s2
            then return {s1, s2};
    end;

    return S;
}
```

What makes two DFA state equivalent?
Is there another dimension for minimization?

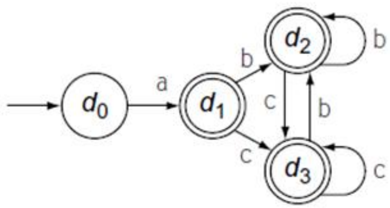
Algorithm from
"Cooper, K.D., Torczon, L.; Engineering A Compiler"



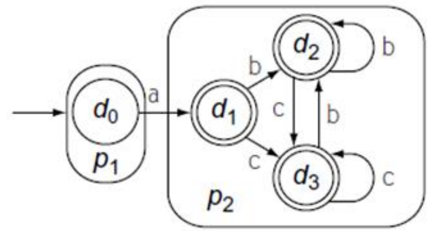
DFA

Minimization

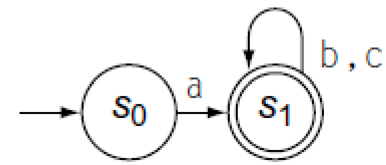
a(b|c)*



(a) Original DFA



(b) Initial Partition



Accept	State	a	b	c
	d_0	1	-	-
*	d_1	-	2	3
*	d_2	-	2	3
*	d_3	-	2	3

Group	Accept	State	a	b	c
0		d_0	1	-	-
1	*	d_1	-	2	3
1	*	d_2	-	2	3
1	*	d_3	-	2	3

Applying Hopcroft's algorithm **g_0={d_0}, g_1={d_1, d_2, d_3}**

Group	Accept	State	a	b	c
0		g_0	1	-	-
1	*	g_1	-	1	1

Applying Symbol Minimization **m0={a}, m1={b, c}**

Group	Accept	State	m0	m1
0		g_0	1	-
1	*	g_1	-	1

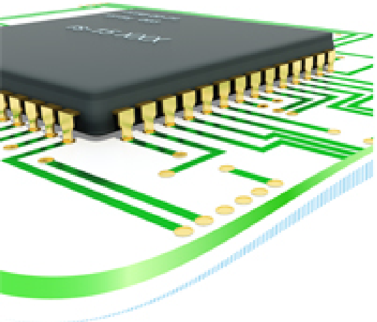
Excerpts from "Cooper, K.D., Torczon, L.; Engineering A Compiler"

Complexities

Build and Recognition

AUTOMATON	INITIAL	PER STRING
NFA	$O(\|r\|)$	$O(\|r\| \times \|x\|)$
DFA typical case	$O(\|r\|^3)$	$O(\|x\|)$
DFA worst case	$O(\|r\|^2 2^{\|r\|})$	$O(\|x\|)$

Table from “Aho, A.V, Ullman J.D, Sethi R., Lam M.S; Dragon Book”



DSA

Elementary structures and Algorithms

- **Set**

  Symbols, States
- **Stack / Queue / List / Array …**

  States
- **Graph, Matrix**

  Transition Functions, Symbols, Sets

- **What about symbols?**

  ANSI Characters

  Unicode

  Case Sensitivity!