Formal Languages

- **Regular expressions**
- **Finite state automata**
  - **Deterministic**
  - **Non-deterministic**
- **Review of BNF**
- **Introduction to Grammars**
  - **Regular grammars**

Formal Languages

- **A way to describe difficulty of computation problems formulated as language recognition problems**
- **A mechanism to aid description of programming language constructs**
  - **Regular expressions - PL tokens (e.g., keywords)**
  - **Finite state automata (FSAs)**

Regular Expressions

- **Formalism for describing simple PL constructs**
  - **reserved words**
  - **identifiers**
  - **numbers**
- **Simplest sort of structure**
- **Recognized by a finite state automaton**
- **Defined recursively**

Formally

- **PL is a set of strings (called *sentences*) over some finite alphabet of symbols, called *terminals***
  - **Not necessarily a finite set**
- **Rules describe how to combine the terminals into well-formed sentences in the PL**
- **PLs categorized by complexity of these rules**
  - **BNF used to describe *context-free languages*, most PLs fall in this category**

Regular Expressions

PL construct	RE Notation	Language
	an empty RE	{ }
symbol a	a	{a}
null symbol	ε	{ε}
R,S regular exprs	R | S	$L_R \quad L_S$
a,b terminals	*a|b (alternation)*	*{a,b}*
R,S regular exprs	RS	$L_R L_S$
a,b terminals	*ab (concatenation)*	*{ab}*

Regular Expressions

PL construct	RE Notation	Language
R,S regular exprs	R^*	{ε } $L_R \quad L_R\, L_R \quad L_R\, L_R\, L_R \ldots$
a	a^*	*{ε ,a,aa,aaa,…}*
R,S regular exprs	R^+	$L_R \quad L_R\, L_R \quad L_R\, L_R\, L_R \ldots$
a	a^+	*{a,aa,aaa,…}*

Note: ε a = a ε = a

Precedence is {* +} ----concatenation ---- |

high to low

(all are left associative operators)

RE Examples

$1 \mid 2$	**$\{1,2\}$**
$1^* \mid 2$	**$\{2, \varepsilon,1,11,111,\ldots\}$**
$1\ 2^*$	**$\{1, 12, 122, 1222, \ldots\}$**
$1\ 2^* \mid 0^+$	**$\{0,00,000,\ldots,1,12,122,\ldots\}$**
$(1 \mid 2)^*$	**$\{\varepsilon,1,2,12,11,21,22,\ldots\}$**
$(0 \mid 1)^*\ 1$	**Binary numbers that end in 1**

RE's for PLs

- **Let *letter* stand for a|b|c|…|z and *digit* stand for 0|1|2|3|4|5|6|7|8|9**
  - ***letter (letter | digit)*** * **is identifier**
  - ***digit*** $^+$ **is integer constant**
  - ***digit*** ***.** ***digit*** $^+$ **is real number**
- **Which identifiers are described by**
  - ***letter (letter | digit)*** * **? ABC 0C B% X1**

Examples

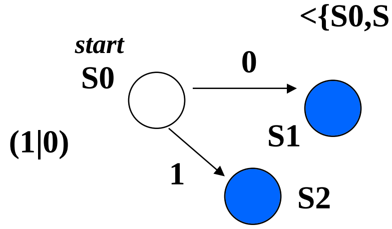
- **Which of the following are legal real numbers described by**
  - **– *digit* * . *digit* $^+$?   .5   1.5   2   4. 6.3 0.2**
- **Can see that simple PL constructs can be defined as regular expressions**
  - **– Can you define a number in scientific notation as an RE?**

Finite State Automaton (FSA)

- **Recognizer of the language generated by a regular expression**
- **Described by**

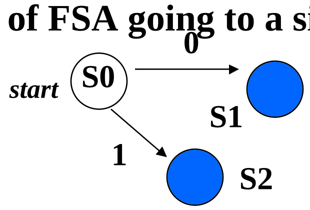
<set of states, labelled transitions, start state, final state(s)>

<{S0,S1,S2}, $S0 \xrightarrow{0} S1$, $S0 \xrightarrow{1} S2$, S0, {S1,S2}>



FSA

- FSA *accepts* or *recognizes* an input string iff there is a path from its start state to a final state such that the labels on the path are the terminals in that string
  - Empty transitions signify illegal moves; can think of FSA going to a sink error state



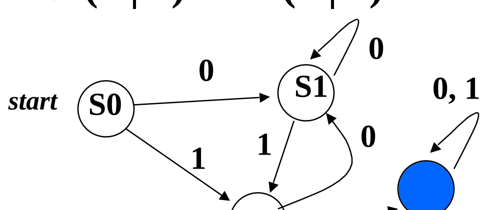
inputs:

states:	0	1
S0	S1	S2
S1	---	---
S2	---	---

transition table

Examples

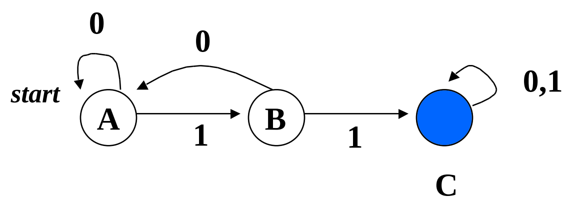
Binary numbers containing a pair of adjacent 1's: (0 | 1) * 1 1 (0 | 1) *



FSA1

	0	1
S0	*S1*	*S2*
S1	*S1*	*S2*
S2	S1	S3
S3	S3	S3

An Equivalent FSA



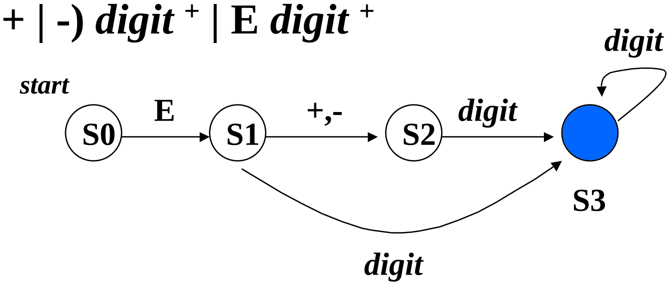
FSA2

FSA1 and FSA2 recognize the same set of strings of terminals, the same language! Therefore **FSAs are NOT UNIQUE.**

Example

Exponent in scientific notation:

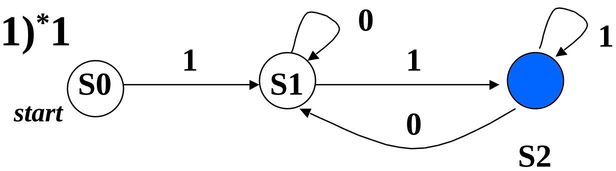
E (+ | -) *digit* $^+$ | E *digit* $^+$



Example

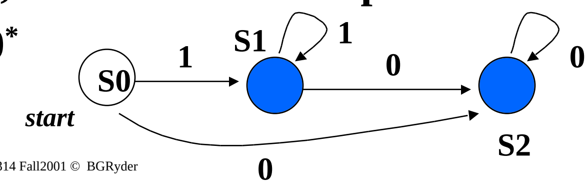
Binary numbers which begin and end with a 1,

$1(0|1)^*1$



All binary numbers containing at least one digit, where all their 1's precede all their 0's

$0^+ \mid 1^+0^*$

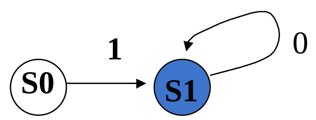


Jobs for REs/FSAs

- **Recognition**
  - **Is this string in the language described (recognized) by this RE (FSA)?**
- **Description**
  - **Given an RE (FSA), what language does it generate (recognize)?**
- **Codification**
  - **Given a language, find an RE and FSA corresponding to it**

Example

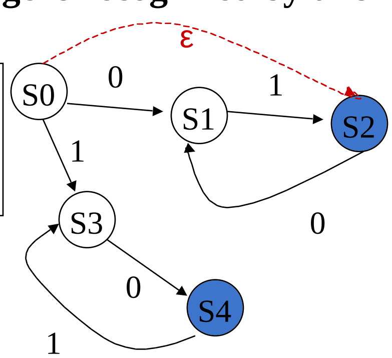
- **Recognition**
  - Given 10*, which of these strings are described by it?   1, 00, 10, 1000, 01
  - Which of these strings 1, 00, 10, 1000, 01 is recognized by the following FSA?



Example

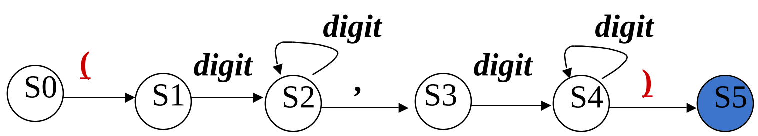
- **Description**
  - What language is generated by $(01)^+ \mid (10)^+$?
  - What language is recognized by this FSA?

What if we added the ε transition?



Example

- **Codification**
  - **Complex constants are parenthesized pairs of two integers**
    - **Let *digit* stand for (0|1|2|3|4|5|6|7|8|9). Then the RE is (*digit* $^+$, *digit* $^+$)**
    - **FSA is:**



Nondeterministic FSAs

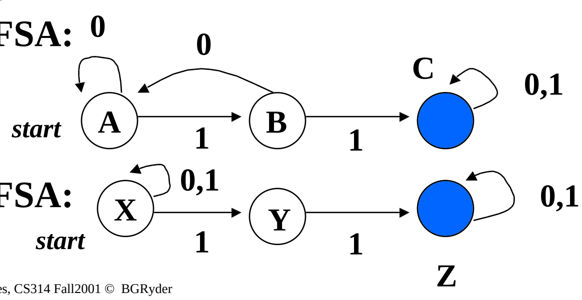
- **Allow more than one transition with same label**
- **Allow ε transition**

---

e.g., (0 | 1) * 1 1 (0 | 1) *

DFSA:

NFSA:



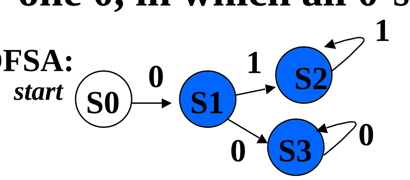
NFSAs

- **Recognize a sentence in a language by progressing from initial state to a final state**
  - **Think of following many threads of computation at same time; one must lead to a final state for a recognition to occur**
- **class of languages recognizable by NFSAs is *SAME* as class of languages recognizable by DFSAs.**
- **There are algorithms to build NFSA directly from RE**

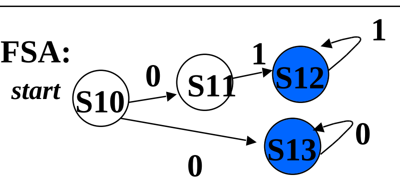
Example

$0^{+} \mid 0\ 1^{+}$ all binary numbers containing at least one 0, in which all 0's proceed all 1's

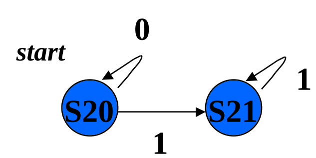
DFSA:



NFSA:



Why doesn't this NFSA work?

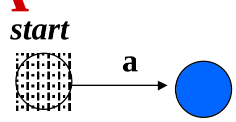


RE to NFSA Construction

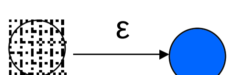
- **Standardized translation for RE expressions into corresponding NFSAs**
- **Can then translate resulting NFSA into a corresponding DFSA which recognizes the same language!**
- **All can be automated**

RE to NFSA

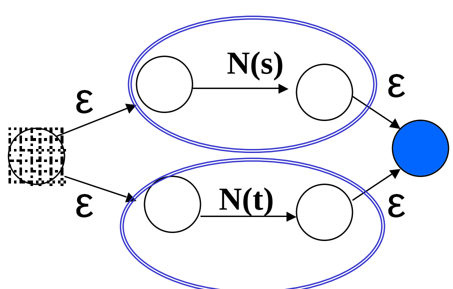
- **For a in alphabet, construct:**

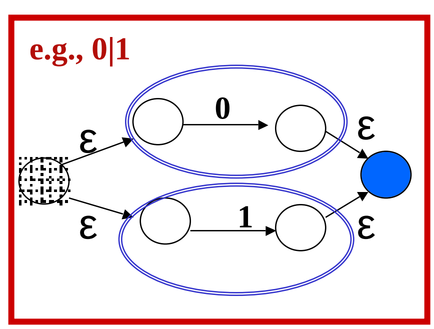


- **For ε, construct:**



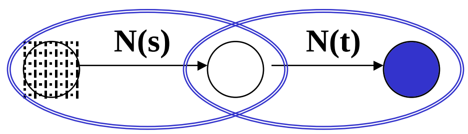
- **For s,t REs, for s|t construct:**

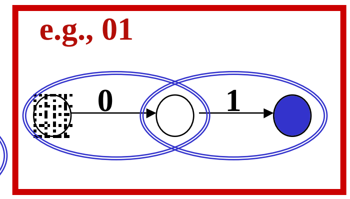




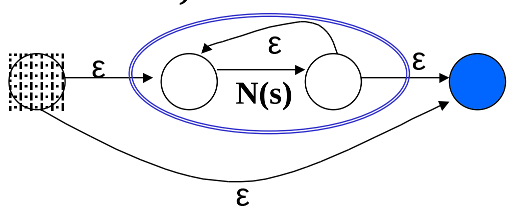
RE to NFSA

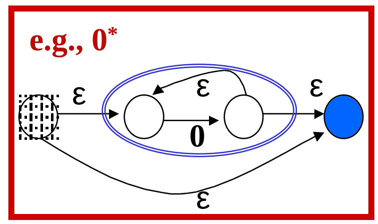
- **For s,t REs, for st construct:**





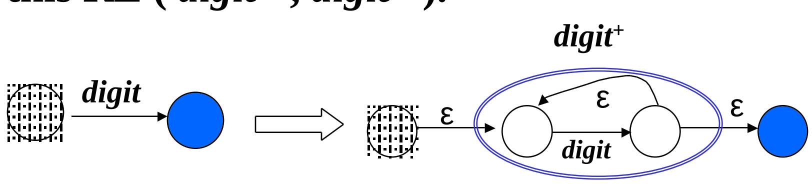
- **For RE s, s* construct:**





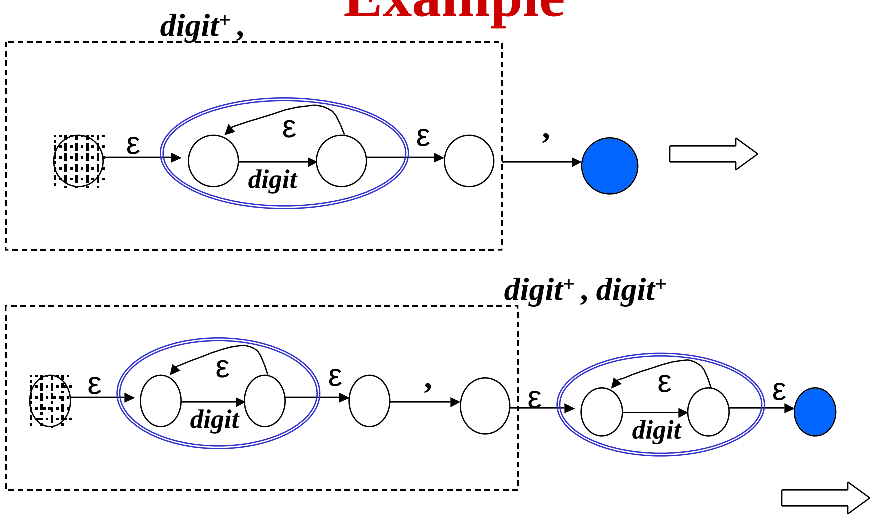
Example

- **Build the NFSA for complex numbers using this RE (*digit* $^+$, *digit* $^+$).**

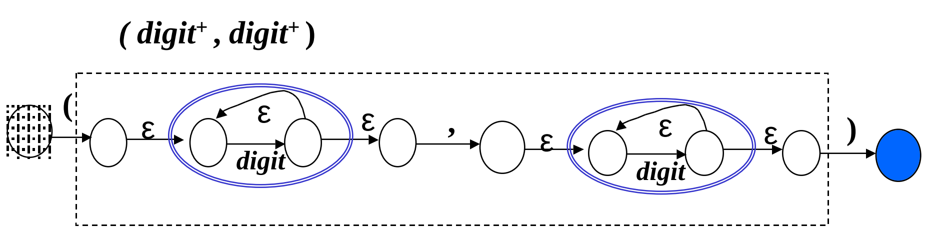


Note this is same as Kleene * machine except for bottom ε transition

Example



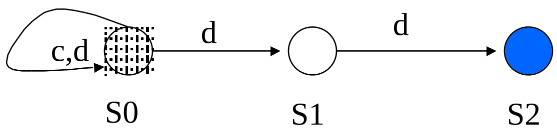
Example



Q: How can we make this NFSA efficient by converting it into a DFSA?

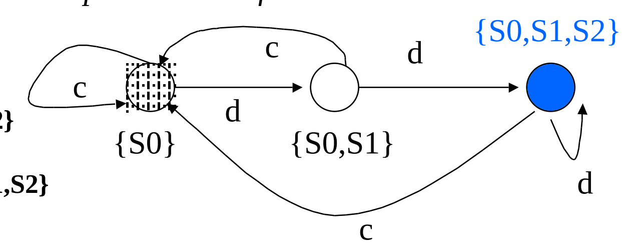
NFSA to DFSA

(c | d)* dd



Idea: look for sets of states with same transitions.
Let one state in the DFSA represent sets of states in the NFSA

S0 on c to {S0}
S0 on d to {S0,S1}
{S0,S1} on c to {S0}
{S0,S1} on d to {S0,S1,S2}
{S0,S1,S2} on c to {S0}
{S0,S1,S2} on d to {S0,S1,S2}



Backus Naur Form (BNF)

- **Metasymbols < > ::= |**
- **Terminal symbols of the PL**
  - **e.g., keywords, operators)**
- **Nonterminal symbols**

<while_stmt> ::= while <expr> do <stmt>

<identifier> ::= <letter> | <identifier> <digit> |
<identifier> <letter>

EBNF

- Nonterminals begin with capital letters or are shown in a different font
- {…} means repeat the enclosed 0 or more times
- […] means the enclosed is optional
- (…) is used for grouping, usually with the alternation symbol |
- If { }, [ ], or ( ) are terminals in the PL being defined, then when they are used as terminals they must be underlined
  - <u>{ }</u> terminals, { } metasymbols

EBNF Examples

Identifier ::= Letter { LetterorDigit }

LetterorDigit ::= Letter | Digit

Expr ::= [Expr -] Subexpr

IfStmt ::= if LogicExpr then Stmt [else Stmt]

CompoundStmt ::= begin Stmt {; Stmt} end

WhileStmt ::= while <u>(</u> LogicExpr <u>)</u> Stmt {; Stmt}

ArrayElement ::= Identifier <u>[</u> Identifier <u>]</u>

Grammar

- <set of terminals, set of nonterminals, productions (rules), special symbol>
  - terminals are alphabet symbols
  - nonterminals represent PL constructs (e.g., Stmt)
  - productions are rules for forming syntactically correct constructs
  - special symbol tells where to start applying the rules

Example

<letter>::=
a|b|c|d|e|f|g|h|i|j|k|l|m|n|o|p|q|r|s|t|u|v|w|x|y|z

<digit>::= 0|1|2|3|4|5|6|7|8|9

<identifier> ::= <letter> | <identifier> <letter> | <identifier> <digit>

<assign-stmt> ::= <identifier> = 0 //terminals;

//nonterminals are {<letter><digit><assign_stmt><identifier>}

//special symbol is <assign-stmt>

Regular PLs

- **Form of rules**
  - **Each rhs is length <= 2 symbols**
    - **A terminal or nonterminal**
    - **a nonterminal followed by a terminal**
- **All PLs describable by REs can be written as regular grammars**

e.g.,1 2^* | 0^+

N::= X | Y

X ::= 1 | X 2

Y ::= 0 | Y 0