```
- Notes on 03-05-18
- References:
```


Review and homework

Going Over Homework

1.1 **Give a regex for strings beginning or ending in A**

```
a[a-z]* | [a-z]*a
```

1.2 **No 3 consecutive B's**

```
(bb|b|ε) (a|ab|abb)*
or
(bba|ba|a)*(bb|b|ε)
```

1.3 **Java 9 Identifiers**

You can have letters or underscore or identifiers

```
(_|ε)[A-A a-z $_][A-z a-z $_ 0-9]*
```

State	A	B	C
A = {0}	B = {1,2,3,4,6,9}	—	—
B (final state)	—	c= {3,4,5,6,8,9}	D = {3,4,6,7,8,9}
C (Final State)	—	C	D
D (Final State)	—	C	D

(order doesn't seem to matter?)

to make it more efficient: we partition it by final states

{A} {B,C,D}

state	a	b	c
A	B	__	__
(B)	__	B	B

Exam Review

- Lexical analysis
- linereal analysis or scanning
- the stream of characters making up the source program is read left to right to create tokens
- Syntax:
    - follow grammar
    - heirarchal
    - make sure program is the right structure
    - also called: heirarchical analysis

- Semantic:
    - make sure they fit together in a meaningful fashion
- Synthesis phases
    - source is translated to intermediate code
    - then make sure to remove redundancies
    - create code generation
- steps (a – d) are the front end of the compiler
- Know these definitions (from blackboard)
    - Compiler(*in lecture 2*)
    - Alphabet (in lecture notes pt. 3 lexical analysis)
        - a finite set of symbols
    - string
        - finite sequence of symbols from a fixed alphabet E
    - language
        - a set of strings over a fixed alphabet
    - concatenation of languages
        - concatenation: combine 2 strings
        - all possible combos of strings from one language to another
        - concat l1,l2 = {st|sε}....?? (Lexical analysis)
    - Kleene closure
        - exponentiation
        - L0 = {ε} Li = Li-1 L (language containing empty string)
        - Union from i=0 to infinity of all your Li's
    - regular expression(still lexical analysis. slide #20 & #21)
        - strings over the alphabet ε + {),(,|,*} such that:
        1. Ø is a regular expression denoting the empty set
        2. ε is a regular expression denoting the set { ε }
        3. a is a regular expression denoting set { a } for any a in E
        4. If r denotes L(r) and s denotes L(s), (regular expressions over E), then:
        - r | s is a regular expression denoting L(r) ∪ L(s) (union)
        - rs is a regular expression denoting L(r)L(s)

(concatenation)
        - r* is a regular expression denoting (L(r))* (closure)
    - pattern
    - lexeme
    - token (Syntax Analysis somewhere around here)
    - context free grammar
        - a 4-tuple G = (N,T,P,S) where:
            - T
            - dude
    - start symbol
    - finite automaton (lexical analysis slide 43)
        - A non deterministic finite automaton is a mathematical model denoted by the 5-tuple...??
        - A Deterministic Finite automatata is a s
    - derivation (module 4 for part 'A') (syntax analysis A #12)
        - A one-step derivation is defined by a,AB => AYb
    - leftmost derivation
    - ambiguous grammar
        - If you have 2 parse trees for the same input string it's ambiguous
- Be able to construct a regular expression given a description of a regular language.
    - Homework 1
- Be able to describe the language a regular expression generates.
    - Inverse of homework 1
- Be able to construct a CFG given a description of a simple context free language.
    - _
- Be able to describe the language a CFG generates.
    - _
- Be able to use Thompson's construction to build an NFA from a regular expression.
    - _
- Be able to use the Subset construction algorithm to convert an NFA

to a DFA.
    - _
- Be able to give an outline of a Lex program, describing each of the three major parts.
    - _
- Be able to write a simple Lex program which will perform some elementary processing on an input file.
    - _
- Be able to write a simple recursive descent parser for a simple language.
    - _
- Be able to remove left recursion (direct or indirect) from a CFG.
    - _
- Be able to use left factoring to remove common prefixes from a CFG.
    - stmt -> if expr then stmt endif | if expr then stmt else stmt endif
        - compressed:
        - stmt -> if expr then stmt opt_close endif | opt_else -> else stmt | ε