Regular Expressions and Languages

Yo-Sub Han
CS, Yonsei University

Overview of Unit

- Regular expressions
- Regular expressions into FAs: "automata construction"
- FAs into regular expressions: state elimination

Regular Expressions

- Many applications require pattern matching
  - look for <a href> tag for links
  - keyword search/replace
- A regular expression is
  - a pattern that defines a set of strings
  - special syntax used to represet a set:
    e.g.: *.c—a set of patterns that end with .c

Regular Expressions vs FAs

- Regular expressions and FAs are equivalent
- Regular expressions are patterns that can be recognized by FAs (and vice versa)

Regular Expressions

- A regular expression defines a set of patterns:
- Regular expressions are useful in unix/linux (and also OSX): `grep`, `awk`, `sed` etc.
- Lots of applications
  - DNA pattern matching
  - Compiler construction
  - Virus detection
  - ...

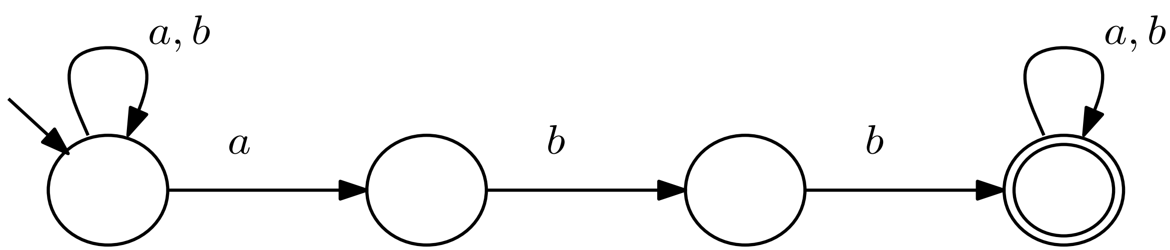
Regular Expression Engines

- Regular expression engines is basically an FA
- A software that can process a text to find regular expression matches
- Regular expression softwares are a part of a larger piece of softwares
  e.g.: `grep`, `awk`, `sed`, `php`, `python`, `perl`, `java` etc
- We can write our own regex engine that recognizes all 'yonsei' in a text
- Different regular expression engines may not be compatible with each other
  e.g.: Perl 5 is a popular one to learn
- One of the best regular expression machines was written in C by Ken Thompson in the 60's
  - superior to `perl`, `python` and other implementations when working with real world applications
  - 400 lines of C code

Regular Expressions

Consider the language L of all strings that consist of a's and b's and have abb as a substring. We can formally define L as follows:

1. $L = \{w \mid w \in \{a, b\}^*$ and w has abb as a substring$\}$
2. $L = L(A)$, where A is an NFA given as follows:



Both definitions are lengthy. It can also be expressed by

$$L((a + b)^* abb(a + b)^*).$$

Regular Expressions

A finite method of specifying/expressing languages

The inductive definition of regular expressions over an alphabet Σ:

1. The $\emptyset$ character, the λ character and each character $\sigma \in \Sigma$ are regular expressions. (Note that $\emptyset$ and λ must not be in Σ.)

2. If α and β are regular expressions, then

$$(\alpha \cdot \beta), (\alpha + \beta) \text{ and } (\alpha^*)$$

are regular expressions (we usually omit "$\cdot$")

For example, given $\Sigma = \{a, b, c, d\}$,

- $a, ((a + b)^* \cdot d), ((c^*) \cdot (a + (b \cdot \lambda)))$ and $\emptyset^*$ are regular expressions
- $c+^*$ and $*$ are not regular expressions

Regular Expressions and Languages

Given two regular expressions E and F,

1. $E+F$ is a regular expression for the union of $L(E)$ and $L(F)$. That is,

$$L(E+F) = L(E) \cup L(F).$$

2. EF is a regular expession of the catenation of $L(E)$ and $L(F)$. That is,

$$L(EF) = L(E)L(F).$$

3. E^* is a regular expression of the closure of $L(E)$. That is

$$L(E^*) = (L(E))^*.$$

4. (E), a parenthesized E, is a regular expression for the same language. That is,

$$L((E)) = L(E).$$

Precedence of Regular Expressions

The regular expression operators have an assumed order of "precedence"; operators are associated with their operands in a particular order.

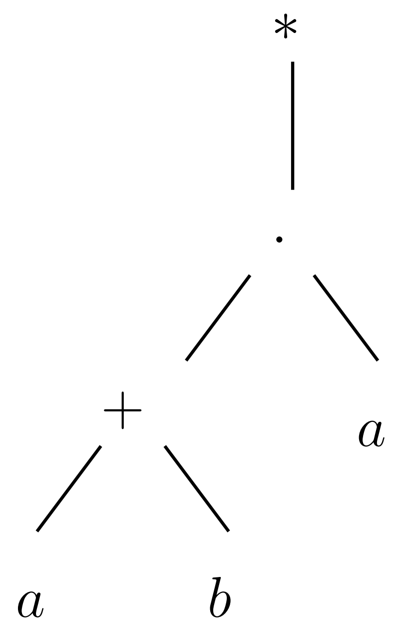
1. The star operator is of highest precedence
2. The catenation operator is next in precedence.
3. The union operator is of lowest precedence

Note that we do not always want the grouping in a regular expression to be as required by the precedence rule. If so, we can use parentheses to group as we want.

$$\{\}, () \rightarrow {}^{*} \rightarrow \cdot \rightarrow +$$

Expression Trees

Given a regular expression E, we can display the *parsing* by an expression tree based on the precedence rule.



$$E = ((a + b)a)^*$$

$$E' = a + ba^*$$

Regular Expression: Algebraic Laws

Given regular expressions E, F and G,

- Commutative law for union: $E + F = F + E$
- Commutative law for catenation: $EF \neq FE$
- Associate law for union: $(E + F) + G = E + (F + G)$
- Associative law for catenation: $(EF)G = E(FG)$
- Left distributive law of catenation over union: $E(F + G) = EF + EG$
- Right distributive law of catenation over union: $(E + F)G = EG + FG$

Regular Expression: Algebraic Laws

Given a regular expression E,

- Identities ($\emptyset$ and λ)

$$\emptyset + E = E + \emptyset = E.$$

$$\lambda E = E\lambda = E.$$

- Annihilator ($\emptyset$)

$$\emptyset E = E\emptyset = \emptyset$$

- Idempotent

$$E + E = E$$

  We define an operator to be idempotent if multiple applications of the operation do not change the result. Note that common arithmetic operators are not idempotent. e.g., $x+x \neq x, x \times x \neq x$

Regular Expression: Algebraic Laws

Given a regular expression E,

- $(E^*)^* = E^*$
- $\emptyset^* = \lambda \neq \emptyset$
- $\lambda^* = \lambda$
- $E^+ = EE^* = E^*E$ (Kleene plus or Plus closure)
- $E^* = E^+ + \lambda$

Regular Languages

We define a language L to be a regular language if and only if there is a regular expression E such that $L = L(E)$. The family of (all) regular languages is denoted by $\mathcal{L}_{REG}$.

Example: Let $E = (b^*ab^*a)^*b^*$ and
$L_{even} = \{w \mid w \in \{a,b\}^* \text{ and } w \text{ has an even number of } a\text{'s; namely, } |w|_a = 2i \text{ for } i \geq 0\}$.

Claim: $L(E) = L_{even}$
Proof:

1. $L(E) \subseteq L_{even}$ since every string in $L(E)$ has an even number of a's
2. Let $w \in L_{even}$. Then, we can write w as

$$w = b^{i_0}ab^{i_1}ab^{i_2}\cdots ab^{i_{2n}} \text{ for } i_0, i_1, \ldots, i_{2n} \geq 0.$$

   This implies that $w = (b^{i_0}ab^{i_1}a)(b^{i_2}ab^{i_3}a)\cdots(b^{i_{2n-2}}ab^{i_{2n-1}}a)b^{i_{2n}}$ and, therefore,

$$w \in L((b^*ab^*a)^*)L(b^*) = L(E).$$

Regular Expressions Example

Given $\Sigma = \{a, b\}$,

1. $L_1 = \{w \mid w = au$ and $u \in \Sigma^*\}$

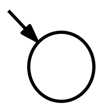
2. $L_2 = \{w \mid |w|_a \equiv 0 \text{ mod } 3\}$

3. $L_3 = \{w \mid w$ has 2 or 3 a's with the last two appearances nonconsecutive$\}$

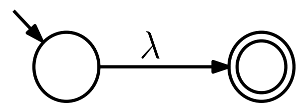
4. $L_4 = \{w \mid w = a^i b^i, i \geq 1\}$

Regular Expressions into FAs

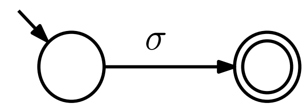
Given a regular expression E over Σ, we can construct a λ-NFA A such that $L(E) = L(A)$ using the following inductive construction:



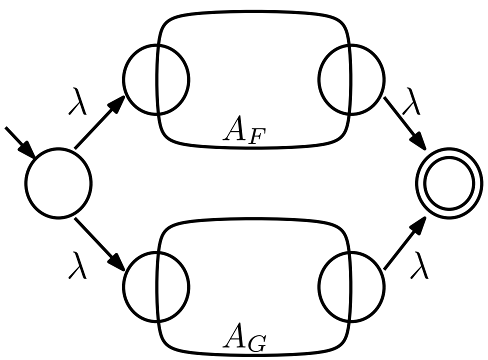
$R_1 : E = \emptyset$
$L(A) = \emptyset$



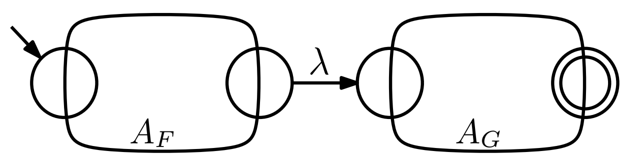
$R_2 : E = \lambda$
$L(A) = \{\lambda\}$



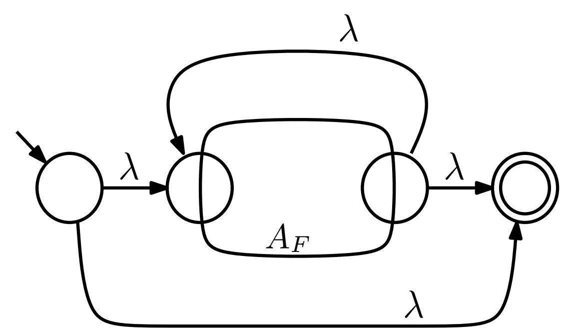
$R_3 : E = \sigma$
$L(A) = \{\sigma\}$



$R_4 : E = F + G$
$L(A) = L(F) \cup L(G)$



$R_5 : E = FG$
$L(A) = L(F)L(G)$



$R_6 : E = F^*$
$L(A) = L(E)$

Thompson Construction Example

We call this FA construction Thompson construction named after the inventor, "Ken Thompson". We can call such FAs *Thompson automata*.

Given $\Sigma = \{a, b\}$,

1. $E_1 = (a + b)^*(\lambda + a)$

2. $E_2 = (a + b^*)^*$

3. $E_3 = (aa + ba)(b^* + a)$

4. $E_4 = b^*(a + ab^*)^*$

Regular Expressions into FAs

Claim: Given E, the Thompson automaton A for E satisfies $L(E) = L(A)$.

Proof: Let $\mathbb{OP}(E)$ be the total number of operators($^*, \cdot, +$) in E. We prove this claim by induction on $\mathbb{OP}(E)$.

Basis: $\mathbb{OP}(E) = 0$. Then, $E = \emptyset, \lambda$ or $\sigma \in \Sigma$ and the claim is true by R_1, R_2 and R_3.
Hypothesis: Assume that the claim holds for all E with $\mathbb{OP}(E) \leq k$ for some $k \geq 0$.
Induction: Consider E such that $\mathbb{OP}(E) = k+1$. Since $k+1 \geq 1$, E must have at least one operator. We have three cases:

1. $E = F + G$. Note that $\mathbb{OP}(F) \leq k$ and $\mathbb{OP}(G) \leq k$. Let A_F and A_G be the corresponding FAs. Then, by the hypothesis, $L(A_F) = L(F)$ and $L(A_G) = L(G)$. Because of R_4, $L(A) = L(A_F) \cup L(A_G)$ and $L(E) = L(F) \cup L(G)$. Therefore, $L(A) = L(E)$.

2. $E = FG$.

   ....

3. $E = F^*$.

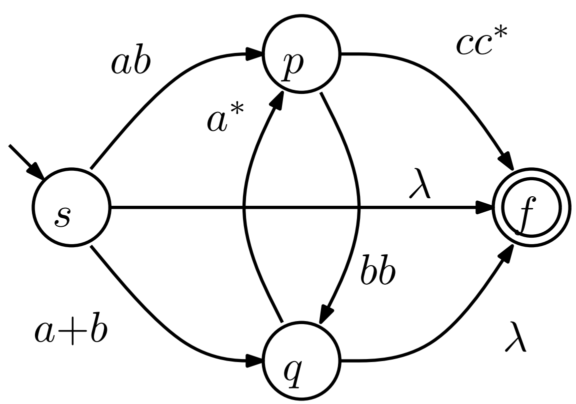
   ....

FAs into Regular Expressions

We now prove that for every FA A, there is a regular expression E such that $L(A) = L(E)$.

An expression automaton (EA): An EA A is an FA with regular expressions as transition labels. Formally, A is specified by a tuple $(Q, \Sigma, \delta, s, f^{\dagger})$, where

1. Q, Σ, s are the same as in λ-NFA
2. s does not have any in-transitions
3. f is the only final state such that $f \neq s$ and f has no out-transitions
4. δ is a set of (Q, R_{Σ}, Q) (in λ-NFA, it is (Q, σ, Q))



†: To be presice, it has to be $\{f\}$. But we use f for short if not confused.

Computations for EAs

Single-step configuration in an EA A:

1. Let (p, w) be a current configuration, where $w = uv$ and $u, v \in \Sigma^*$
2. $(p, E, q) \in \delta$ and $u \in L(E)$, where E is a regular expression
3. We say that (p, w) yields (q, v) in one step. Namely, $(p, w) \vdash (q, v)$
4. We can define $\vdash^*, \vdash^+$ in a similar way: multiple-step configuration

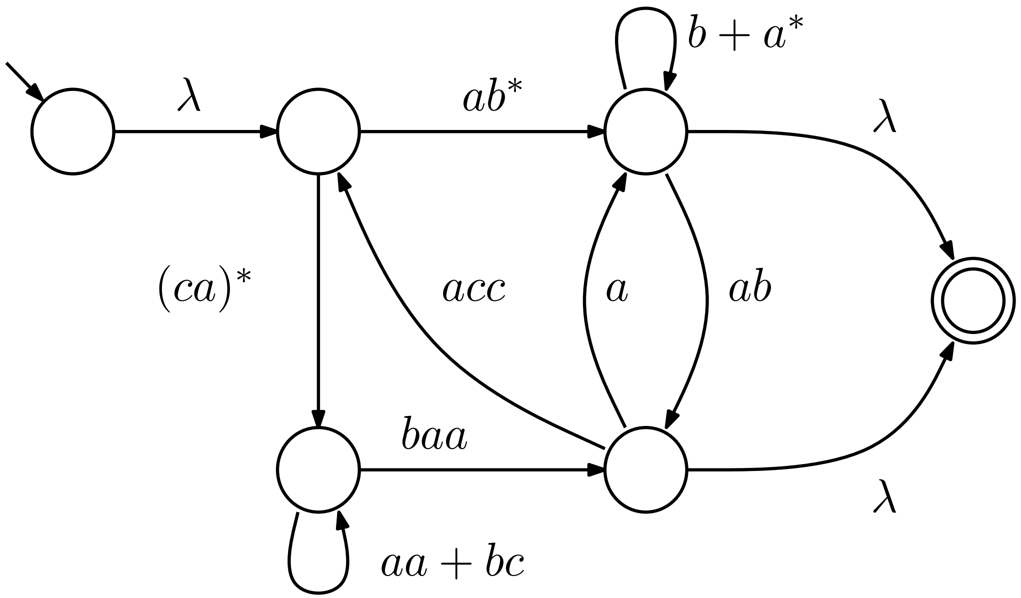
In other words, a transition (p, E, q) is applied to A if p is the current state and there is a string $u \in L(E)$ such that u is a prefix of the unread portion of w of the input string, then A moves into the next state q and the reader consumes u leaving v as the unread input.

Nondeterminism and Acceptance for EAs

Given an EA A and its transition (p, E, q) with the unread input $w = uv$:

1. There may be many strings that satisfy the condition, so A may be highly nondeterministic!
2. If $E = a^*$ and $w = a^k b$, say, where $k \gg 0$, then $u = \lambda, a, aa, aaa, \ldots, a^k$, are all possible choices
3. We define acceptance as we did for normal NFAs:
   A string w is accpetyed by an EA A if there is a computation for w that begins at the start state and ends at the final state such that w has been completely consumed.

Nondeterminism and Acceptance for EAs Example



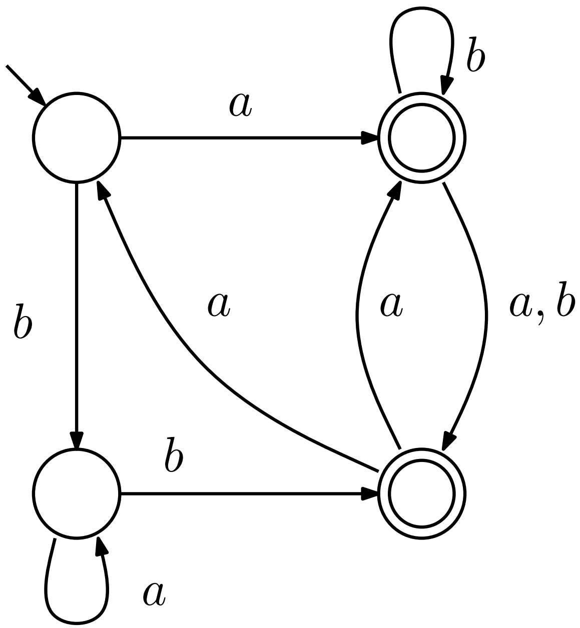
Determine if the following strings are in $L(A)$.

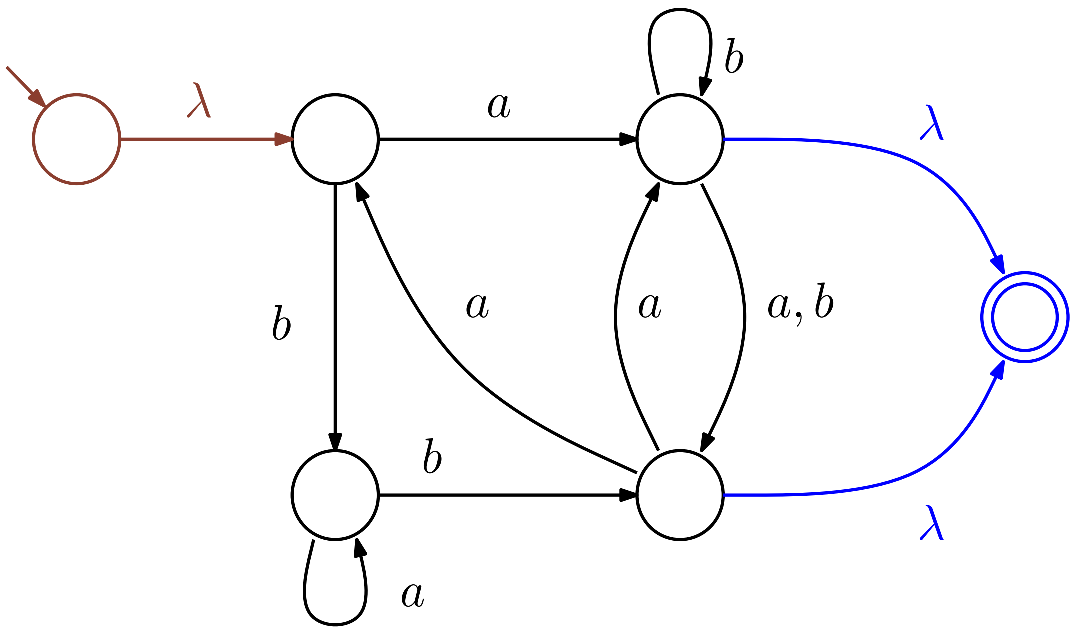
1. $caaabcbaa$
2. $aaaab$
3. baa
4. ac

FAs and EAs

Claim: Given an FA A, there is an EA A' such that $L(A) = L(A')$

Proof: It is left for an exercise. Here is an intuition:

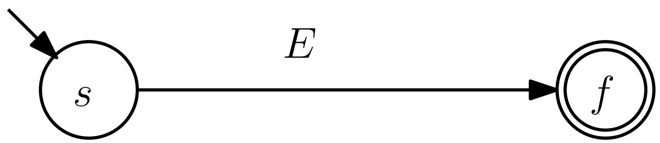




State Elimination

We are now ready to work on state elimination, a very simple idea for computing a regular expression from an FA. At each step, we bypass a nonstart, nonfinal state to give an equivalent automaton (which will be an EA) that has one less state.

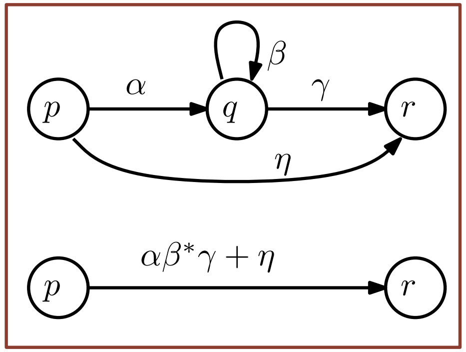
Goal of the technique:



State Elimination

How does state elimination work?

1. First, consider a state q, which we wish to eliminate, that has an in-transition (p, α, q), a self-looping transition (q, β, q) and an out-transition (q, γ, r). (It may have other in/out-transitions.)



2. When we eliminate q, we replace the transition sequence

$$(p, \alpha, q), (q, \beta, q), \ldots, (q, \beta, q), (q, \gamma, r)$$

by

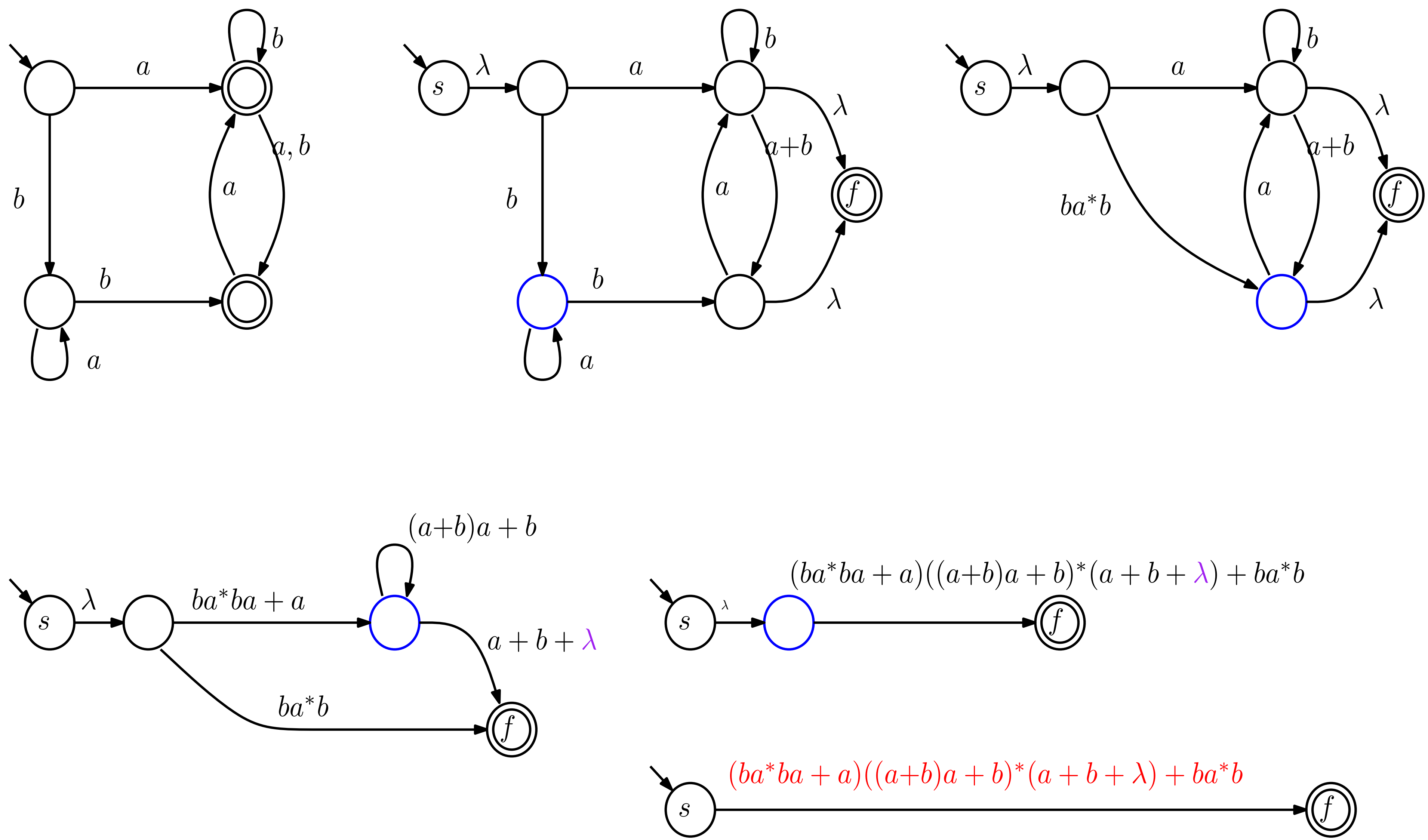
$$(p, \alpha\beta^*\gamma, r)$$

and, since q is not final, we can see that this new transition emulates the previous transition sequence

3. Finally, we union the new expression and the original expression η between p and r: $(p, \alpha\beta^*\gamma + \eta, r)$

4. This observation holds for all shortcuts that we have made to avoid q, so we no longer need q after we have bypassed it

State Elimination Example



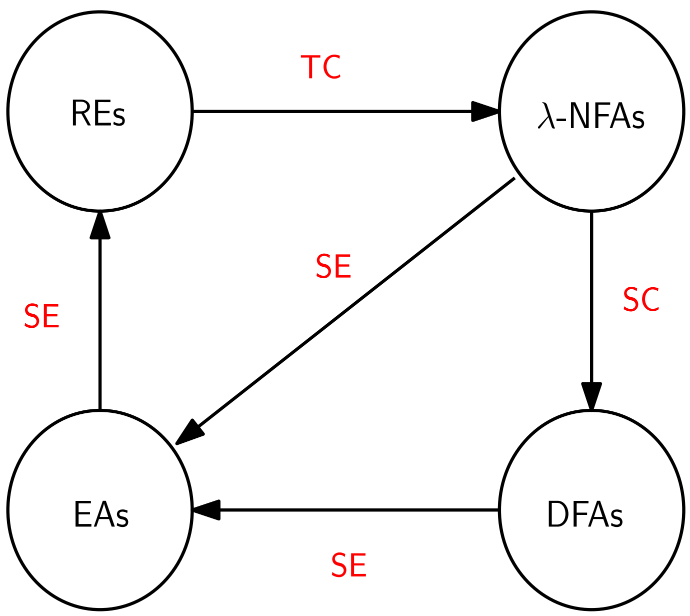
Summary of State Elimination

1. Add a new start state if the original one has an in-transition
2. Add a new final state if there are more than one final states originally or if there is a single final state but it has an out-transition. Old final states become nonfinal states.
3. Eliminate states in $Q \setminus \{s, f\}$ one by one

Summary of State Elimination

From state elimination, we know that

1. Given an FA A, we can compute a regular expression E such that $L(A) = L(E)$ using state elimination
2. EAs have the same expressive power as FAs
3. Both regular expressions and FAs define the same set of languages, regular langaugeses



Applications of Regular Expressions

Regular expressions in UNIX

- extended regular expressions—have additional features
- allow to write character classes to represent large sets of characters as succinctly as possible
  - {0, 1} vs {a,b,c,d,e, . . . , y, z}

The rules for character classes are

- The symbol . (dot): "any character"
- The sequence $[a_1 a_2 \cdots a_k]$: $a_1 + a_2 + \cdots + a_k$
- A range of the form x–y: all the chracters from x to y in the ASCII sequences
  - [0–9]
  - [A–Z]
  - [0–9a–zA–Z]

Applications of Regular Expressions

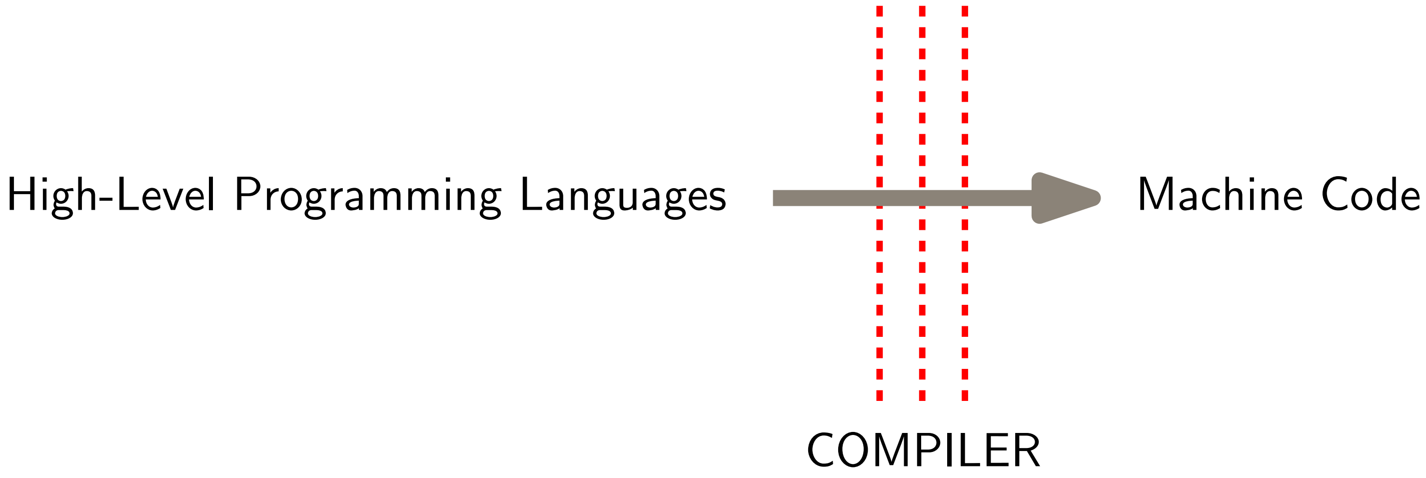
The most common classes of characters

- [:digit:] is the set of ten digits, the same as [0–9]
- [:alpha:] is the set of any alphabetic character, the same as [A–Za–z]
- [:alnum:] stands for the digits and letters (alphabetic and numeric characters), the same as [A–Za–z0–9]

Several UNIX operators

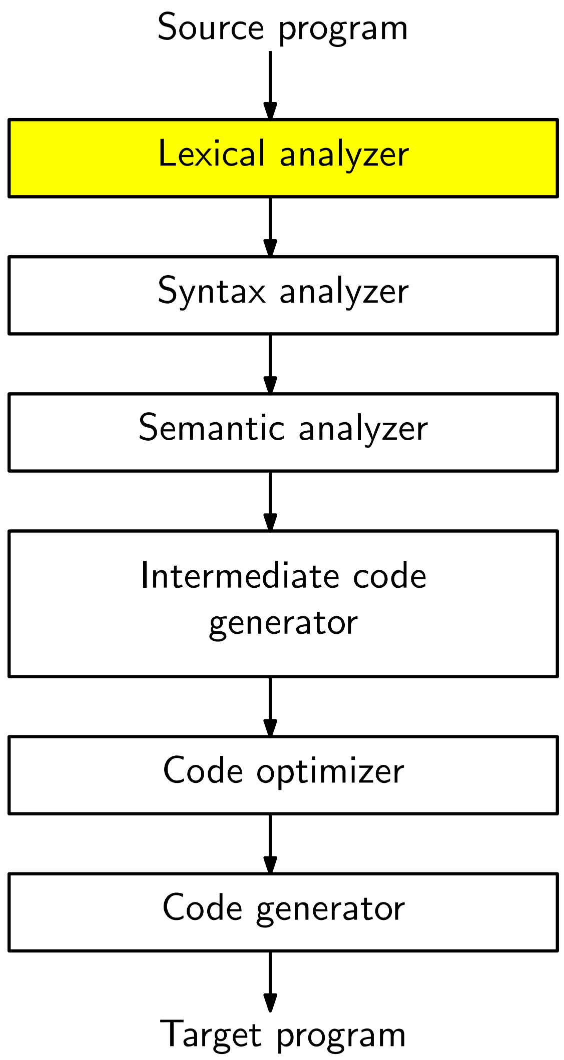
- The operator $|$ is used to denote union $(+)$
- The operator ? means "zero or one of." e.g.: $R? = \lambda + R$
- The operator $\{n\}$ means " n copies of.". e.g.: R$\{3\}$ $=$ shorhand for RRR

Application: Lexical Analyzer in Compiler



1. A program that translates a program in one language to another language
2. The essential interface between applications and architectures

Application: Lexical Analyzer



Lexical analyzer

1. A scanner groups sequence of characters into tokens—smallest meaningful entity in a language (keywords, identifiers or constants)
2. Makes use of regular languages and FAs

Application: Lexical Analyzer

A scanner

1. Recognizes the keywords of the languages (these are the reserved words that have a special meaning such as *if*, *else* or *switch* in C)
2. Recognizes special characters such as (and) or groups of special characters such as := and ==
3. Recognizes identifiers, integers, reals, decimals, strings, etc
4. Ignores whitespaces (tabs and blanks) and comments
5. Recognizes and processes special directives (such as the #include "file" directive in C) and macros

Yo-Sub Han

Application: Lexical Analyzer

An example of some tokens:

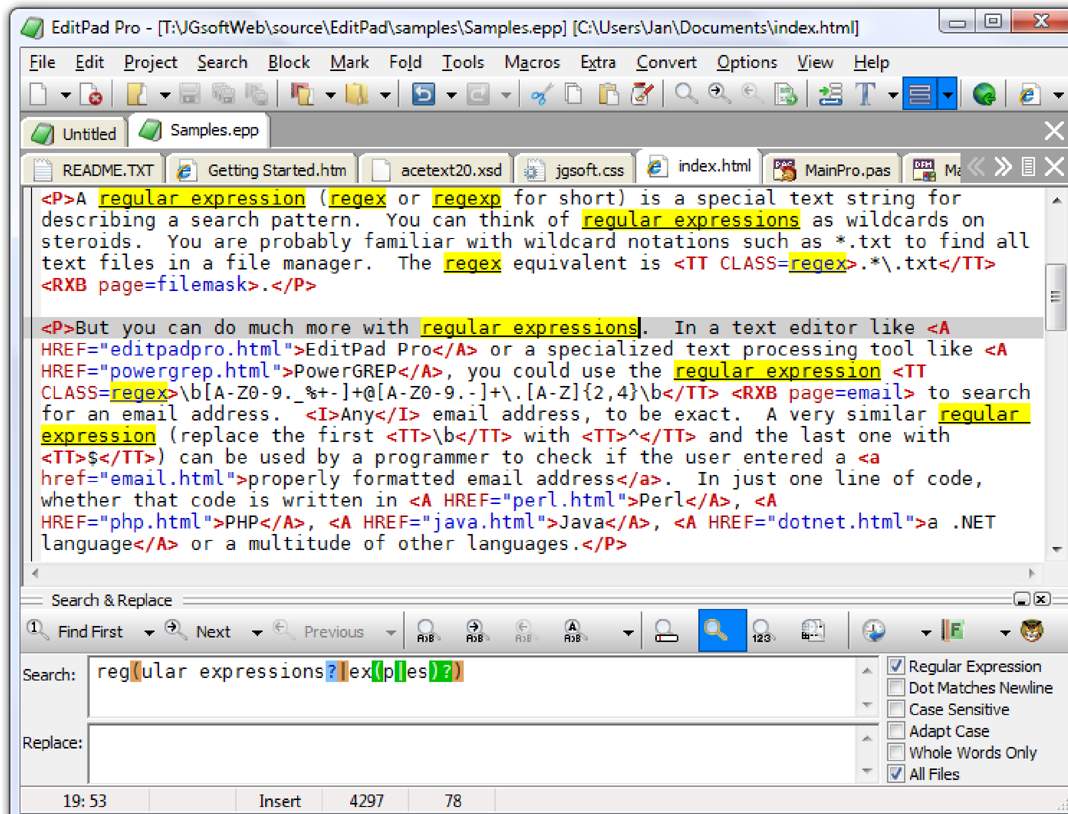
for-keyword	=	for
letter	=	[a-zA-Z]
digit	=	[0 - 9]
identifier	=	letter (letter $+$ digit)*
sign	=	$+ \mid - \mid \lambda$
integer	=	sign (0 $+$ [1 - 9]digit*)
decimal	=	integer . digit*
real	=	(integer $+$ decimal) E sign digit$^+$

Application: Finding Patterns in Text

Bio sequences such as DNA, amino-acid sequences often have regular patterns.

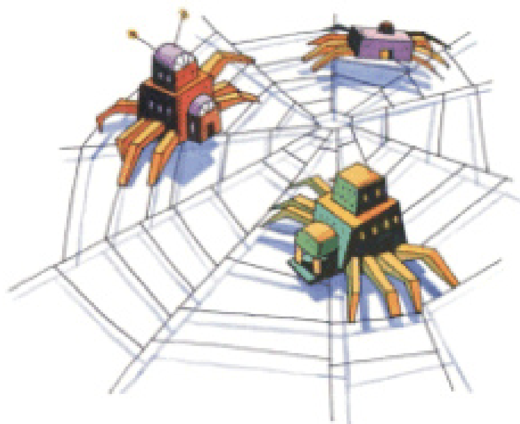
$ATATATGC$
$ATACCTGC$
$ATAGGTGC$
$ATACGTGC$

$$ATA[A \mid T \mid G \mid C]^2TGC$$



Application: Finding Patterns in Text

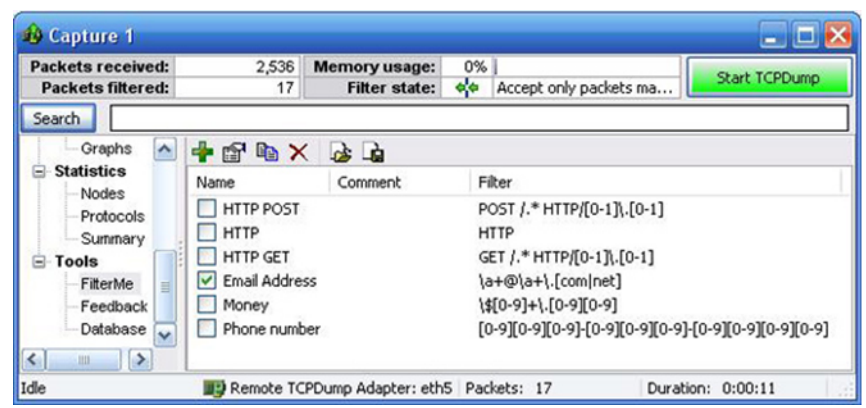
New Domains: WEB, Bioinformatics, Intrusion Detection and so on



Application: Finding Patterns in Text

- Text Search: grep in UNIX/LINUX, preg_match in php, matches in java
- Network Intrusion Detection: Snort www.snort.org
- Antivirus: ClamAV www.clamav.net
- Bioinformatics: PHI-BLAST blast.ncbi.nlm.nih.gov/Blast.cgi

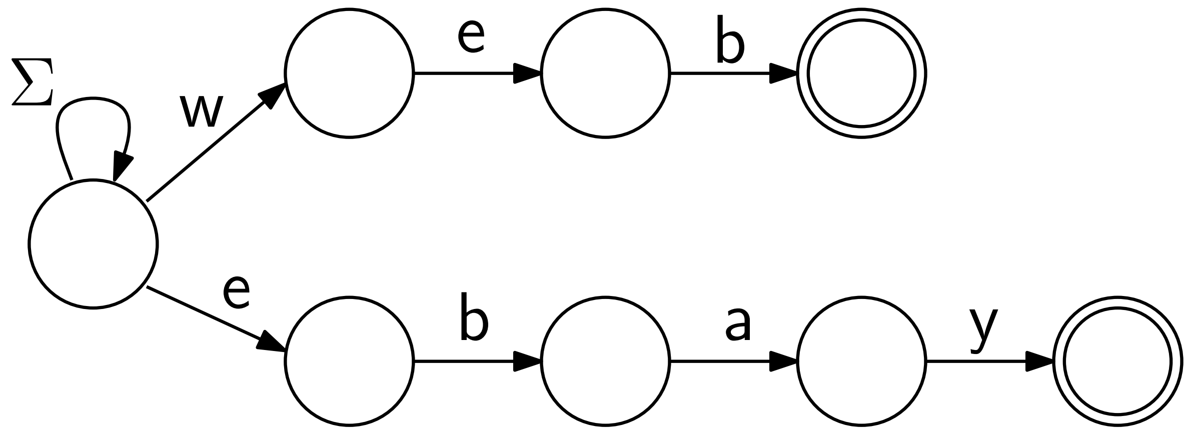
Grep in Unix



Exported Snort rules

Application: Finding Patterns in Text

- Often use NFAs for finding keywords in text
- An NFA to recognize occurrences of the words `web` and `ebay`



Application: Finding Patterns in Text

T $= AGCTAA\boxed{TCCCT}GAGAG\boxed{TCCAGT}\boxed{TAGT}CCCAT$

P $= T \cdot (AG + C)^* \cdot T$