Theoretical Computer Science Summary 2015

Pieter Schaap

March 30, 2015

Contents

1 Lecture 1: Intro

1.1 Languages And Strings

- Lexical analysis: Scan program + break up in variable names, numbers etc.
- Parsing: create tree corresponding to sequence of operations that should be executed.
- Optimization: Skip actions and assignments that are never used.
- Termination: Check if program finishes and does not loop infinitely.
- Interpretation: Figure out what useful it does.
- **Language:** possibly infinite set of finite length strings over a finite alphabet.

- **String:** finite sequence, possibly empty, or symbols drawn from some alphabet $\sum$.
- ϵ is the empty string.
- Σ^* is the set of all possible strings over an alphabet Σ.

1.1.1 Functions on Strings

- **Counting:** $|s|$ = number of symbols in s.
  $|\epsilon| = 0$
- **Concatenation:** st (sometimes $s||t$)) is the concatenation of s and t
- **Replication:** basically copying the string:
  $w^0 = \epsilon$ $w^1 = w$ $w^2 = ww$ $w^{i+1} = w^i w$
- **Reverse:** For each string w, w^R is defined as:
  if $|w| = 0$ then $w^R = w = \epsilon$ if $|w| \geq 1$ then:
  $\exists a \in \Sigma(\exists u \in \Sigma^*(w = ua))$.
  So define $w^R = a\ u^R$
  Or in other words just flip around the string.

Theorem: if w and x are strings, then $(wx)^R = x^R w^R$
example: $(nametag)^R = (tag^R)(name)^R = gateman$
note: proof is available in slides

1.1.2 Relations on Strings

substring: aaa is a substring of aaabbbaaa.
aaaaaa is nota substring of aaabbbaaa.
aaa is a **proper substring** of aaabbbaaa. meaning that aaa is a substring of aaabbbaaa AND they are not equal to each other.
Every substring is a substring of itself. ϵ is a substring of every string.

Prefix: s is a prefix of t iff: $\exists x \in \Sigma^*(t = sx)$.
s is a **proper prefix** of t iff: s is a prefix of t and $s \neq t$

Suffix: s is a **suffix** of t iff: $\exists x \in \Sigma^*(t = xs)$ s us a **proper suffix** of t iff: s is a suffix of t and $s \neq t$. Every string is a suffix of itself.
ϵ is a suffix of every string.

1.2 Defining a language

note: $\emptyset \neq \{\epsilon\}$; the empty set has no strings at all, whereas the other language has just the empty string. note: ϵ does not have to be part of a language.
the language Σ^* contains an infinite number of strings.

example: $L = \{x \in \{a, b\}^* : \text{all a's precede all b's}\}$

example: $L = \{x : \exists y \in \{a, b\}^* : x = ya\}$ means $\{x \in \{a, b\}^* : \text{all x's ending with a }\}$

1.2.1 Perils of using English

$L = \{x\#y : x, y \in \{0,1,2,3,4,5,6,7,8,9\}^*$ and, when x and y are viewed as the decimal representations of natural numbers, square(x) = y $\}$ e.g. 3#9 and 12#144 are in L and 3#8, 12 and 12#12#12 are not in L.

1.3 Enumeration

- Arbitrary Order
- Lexicographic Order (more useful)

1.4 Languages

1.4.1 Size of a language

The smallest language over any Σ is $\emptyset$, with cardinality 0.
Theorem: if $\Sigma \neq \Sigma^*$ is countably infinite. note: proof available in slides.
Theorem: if $\Sigma \neq \emptyset$ then the set of languages over Σ is uncountably infinite.

1.4.2 Functions on Languages

- Set operations
  - Union
  - Intersection
  - Difference
  - Complement
- Language operations
  - Concatenation
  - Kleene star

Concatenation: L_1 and L_2 are languages over Σ
$L_1L_2 = \{w \in \Sigma^* : \exists \in L_1(\exists t \in L_2(w = st))$

Example:

$L_1 = \{cat, dog\}$

$L_2 = \{apple, pear\}$

$L_1L_2 = \{catapple, catpear, dogapple, dogpear\}$

note: $L\{\epsilon\} = \{\epsilon\}L = L$.
note: $L\emptyset = \emptyset L = \emptyset$.

Concatenating languages using variables: The scope of any variable used in an expression that invokes replication will be take to be the entire expression. e.g.:
$L_1 = \{a^n : n \geq 0\}$,
$L_2 = \{b^n : n \geq 0\}$
then:
$L_1 L_2 = \{a^n b^m : n, m \geq 0\}$ note: NOT $a^n b^n$!!

Kleene Star: $L^* = \{\epsilon\} \bigcup \{w \in \Sigma^* : \exists k \geq 1 (\exists w_1, w_2, ...w_k \in L(w = w_1 w_2 ... w_k))\}$

Example: L = {dog, cat, fish }
then:
L^* = { dog, cat, fish, dogdog, dogcat, fishcatfish, fishdogdogfishcat,... }. So it basically consists of all possible combinations of symbols defined in L.

the $^+ = LL^*$
$L^+ = L^* - \{\epsilon\} iff \epsilon notin L$

L^+ is the closure of L under concatenation.

1.5 Semantic Interpretation Functions

Assigns meanings to strings of a language.

2 Lecture 2: Language Hierarchy

Decision problem: problem for which the answer is yes or no. **Procedure:** answers a decision problem. This is done by solving a language recognition problem: given a language L and a string w; is w in L?

2.1 power of encoding

computers: everything is binary string.

Problems can be encoded (cast) into new problems that are decision problems.

For example given a search string w and a web document d, do they match? in other words should a search engine on input w, consider returning d? The language to be decided: $\{< w, d >: disacandidatematchforthequeryw\}$

Note: there are more examples on the slides!

2.1.1 Turning problems into decision problems

- **Multiplication Integers:** Basically we transform computing into verification, see which string is in the language in order to find out what the answer is:
  $L = \{woftheform :< integer_1 > x < integer_2 >=< integer_3 >, where :< integer_n > isanywellformedinteger, andinteger_3 = integer_1 * integer_2\}$

- **Sorting Integers:** Transform sorting into examining a pair of lists:
  $L = \{w_1 \# w_2 : \exists n \geq 1 (w_1 is of the form < int_1, int_2, ..., int_n >, w_2 is of the form < int_1, int2, ...int_n >, and w_2 contains the same objects as w_1 and w_2 is sorted)\}$.

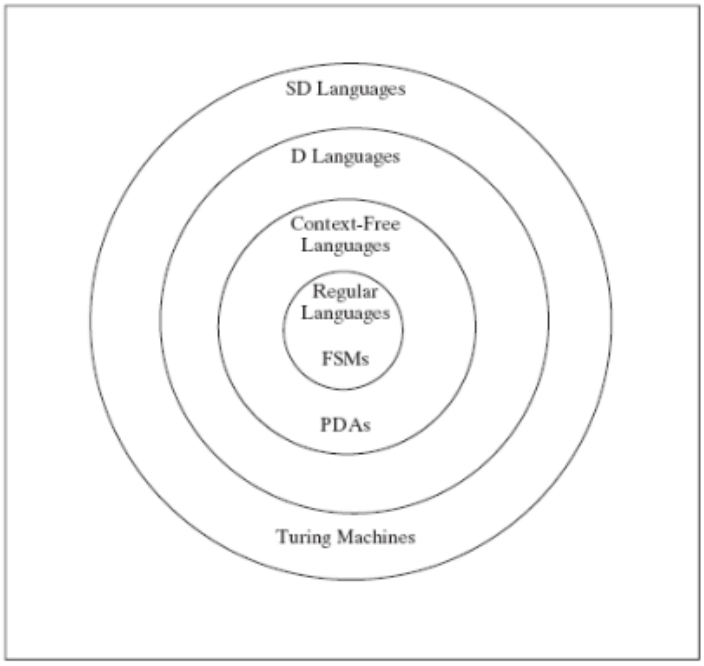
- **DB Querying:** transform query execution into evaluating reply for correctness:
  $L = \{d \# q \# a : d$ *is an encoding of a database,* q *is a string representing a query, and* a *is the correct result of applying* q *to* $d\}$

2.2 Equivalence of problems

equivalent: either problem can be **reduced to** the other. i.e. a machine to solve one problem we can use it to solve the other using just the starting machine and other functions that can be build using a machine of equal or lesser power.

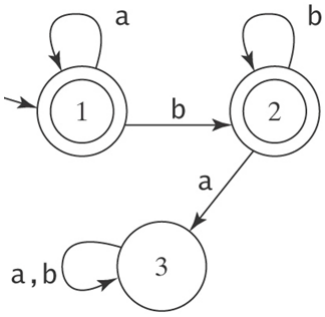
2.3 Machines

Rule of least power: Use the least powerful language suitable for expressing information, constraints or programs on the world wide web.



2.3.1 Finite State Machines

An FSM to accept: $L = \{a^*b^*$ where all a's come before all b's $\}$



note: double circle means accepting state, single circle means rejecting state.

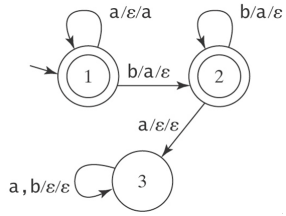
FSM's accept **regular languages**. For example an FSM can not handle $A^nB^n = \{a^nb^n : n \geq 0\}$.

2.3.2 Pushdown Automata

Basically a FSM with a single stack. a label x/y/z means: transition if x is read and y can be popped from stack; then poop y and push z; if y = ϵ, don't pop anything; if z = ϵ, don't push anything.

If all input is read and stack is empty and in accepted state, the input string is accepted, otherwise it is rejected.

A PDA to accept $A^nB^n = \{a^nb^n : n \geq 0\}$:



PDA's accept **context-free languages.** For example A^nB^n or BAL, language of balanced parentheses, e.g. (()), ()() is also context-free. Most programming query and markup languages are context-free.

However, $A^nB^nC^n = \{a^nb^nc^n : n \geq 0\}$ can not be dealt with by PDA's. (thus not context-free)

2.3.3 Turing Machines

Details follow later in the lecture on Turing machines.

Turing machines accepts the classes of **decidable** and **semi-decidable languages.**

- **decidable**: there exists a Turing machine M that halts on all inputs, accepts all strings in L and rejects all strings not in L.

- **semi-decidable**: there exists a Turing machine M that accepts all strings in L and fails to accept all strings not in L (either rejects or loops forever).

2.4 Complexity

Part V: complexity: if decidable in principle, how about resources?

- **P**: polynomial time with length of input
- **NP** languages decided by non-deterministic machine where one path can be explored in time growing polynomially with the length of the input (but the whole tree possibly exponentially)
- **PSPACE** space consumption growing polynomially with the length of the input.

$P \subseteq NP \subseteq PSPACE$

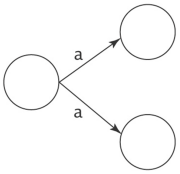
2.5 Computation

2.5.1 Decision Procedures

A decision procedure is basically a yes or no question that is to be answered. It does not necessarily have to be deterministic (i think).

2.5.2 Nondeterminism

Sometimes an action can have more than 1 possible outcome. This means that for some action/input a there are 2 possible states:



Note that in order for a string to be accepted only 1 of the possible paths has to accept the string!

2.6 Functions on Languages

2.6.1 maxstring(L)

maxstring(L) = $\{w \in L : \forall z \in \Sigma^*(z \neq \epsilon \rightarrow wz \notin L)\}$

Examples:

- maxString(A^nB^n) = A^nB^n with $n \geq 0$. maxstring(a^*) = $\emptyset$

All **finite** languages are closed under maxstring because $|maxstring(L)| \leq |L|$
The set of **infinite** languages is not closed, see the a^* example as a counter example!

3 Lecture 3: Finite State Machines

3.1 Deterministic Finite State Machines (DFSM)

3.1.1 Definition

Defined by a 5 tuple: M = $(K, \Sigma, \delta, s, A)$

- K is a finite set of states
- Σ is an alphabet
- $s \in K$ is the initial state
- $A \subseteq Kisthesetofacceptingstaes.$
- δ is transition function from $(Kx\Sigma)$ to K.

3.1.2 Accepting language

Essentially: accept string w iff M winds up in some element of A when it has finished reading w. **Accept:** string w iff: (s,w) $\vdash_m^*$ (q, ϵ) for some $q \in A_m$. **Reject:** string w iff: (s,w) $\vdash_m^*$ (q, ϵ) for some $q \notin A_m$.

3.1.3 Configurations

Configuration: element of K x Σ^*, contains current state and input still left to read. **Initial configuration** is (s_M, w)

3.1.4 Relations

- Yields-in-one-step relation $\dashv_M$ (q',w') iff w . $\dashv_M^*$ is the reflexive, transitive closure of $\dashv_M$ (basically means that by using any combination of $\dashv_M$ operations one can deduct that.. (right hand side).

3.1.5 Computations

Computation: Finite sequence of configurations $C_0, C_1, ..., C_n$ for some n $\geq$ 0 s.t.:

- C_0 is the initial configuration (the starting point/state)
- C_n is of the form (q, ϵ), for some state $q \in K_M$
- and lastly: $C_0 \dashv_M C_1 \dashv_M C_2 \dashv_M ... \dashv_M C_n$. i.e. $C_0 \dashv^* C_n$

3.1.6 Accepting and Rejecting

- **Accepting:** iff (s,w) $\vdash_M^* (q, \epsilon)$, for some $q \in A_M$.
- **Rejecting:** iff (s,w) $\vdash_M^* (q, \epsilon)$, for some $q \notin A_M$.

If a language is accepted by a Machine M it is denoted as L(M) and is basically the set of all strings accepted by M.
Theorem: Every DFSM M, on input w, halts in —w— steps.

3.1.7 Dead States

Basically is a state where no matter what input follows it will always be rejected. Dead states are often represented with a d in a DFSM diagram. Sometimes dead states are omitted. Meaning that if in a diagram an input is not represented in the diagram it is assumed as being a dead state.

3.1.8 Programming FSMs

- Tip 1: **Cluster** strings that share a "future".
- Tip 2: IF you **create a FSM for** $/L$ you can swap the accepting and rejecting states (accepting becomes rejecting and vice versa). This can be a good way to start if you are having trouble.

3.2 Regular Languages

Regular Expression defines **Regular Language**.
Finite State Machine accepts **Regular Language**.

3.3 Non Deterministic Finite State Machines

3.3.1 Definition

NDFSM is defined by 5-tuple: M = $(K, \Sigma, \Delta, s, A)$ where

- K is a finite set of states
- Σ is an alphabet
- $s \in K$ is the initial state.
- $A \subseteq K$ is the set of accepting states.
- Δ is the transition relation, it is a finite subset of $(Kx(\Sigma \bigcup \{\epsilon\}))xK$

3.3.2 Analyzing NDFSM

You have to find only 1 accepting path. If such a path exists; a string is accepted by that NDFSM. 2 approaches: either explore search tree in depth first manner OR follow all paths in parallel, considering sets of possible states (use your fingers or coins).

Dealing with ϵ transitions: use the eps function: eps(q) is the eps function for state q. it is defined as: $eps(q) = \{p \in K : (q, w) \vdash_M^* (p, w)\}$ An algorithm to compute eps(q):

Data: State q
Result: A list of all states that can be reached from q using only epsilon transitoins
result = { q};
while $\exists p \in$ *result and* $\exists r \notin$ *result and some transition* $(p, \epsilon, r) \in \Delta$ **do**
  insert r into result;
end
return result;

Algorithm 1: eps(q) : get all states reachable using only epsilon transitions

This should be fairly clear; else you can probably figure out yourselves how to get all reachable states.

3.3.3 Simulating a NDFSM

```
   Data: M: NDFSM, w: String
   Result: Wether or not the NDFSM accepts input string w
 1 current-state = eps(s);
 2 while w still has unread characters do
 3 |   c = get-next-character(w);
 4 |   next-state = ∅ ;
   |   /* For every state in our current state list            */
 5 |   foreach state q ∈ current-state do
 6 |   |   foreach state p s.t. (q, c, p) ∈ Δ do
   |   |   |   /* take every state that can be reached by a
   |   |   |      transition for input c and add it to the list of
   |   |   |      next-states                                   */
 7 |   |   |   next-state = next-state ∪ eps(p);
 8 |   |   end
 9 |   end
10 |   current-state = next-state;
   |   /* Take next-state as the new current-state             */
11 end
12 if current-state contains any states in A then
13 |   return accept;
14 else
15 |   return reject;
16 end
```

Algorithm 2: ndfsmSimulate

So in short again what ndfsmSimulate does; it basically follows every path possible to reach for the input string. (It follows these paths simultaneously i.e. some sort of breadth first search).

3.4 NDFSM vs DFSM

- { languages accepted by a DFSM } $\subseteq$ {Languages accepted by a NDFSM}.
- for each NDFSM there is an equivalent DFSM and therefore:
  { Languages accepted by a DFSM } = { Languages accepted by a NDFSM}

3.5 NDFSM to DFSM

```
   Data: NDFSM M
   Result: A DFSM representing the same language as M
   /* Start by computing eps(q) for all states in the NDFSM M */
 1 foreach q ∈ K_M do
 2 |   Compute eps(q) /* Alternatively you can compute it on the
   |      go if you are doing this by hand                    */
 3 end
   /* Now we can actually start by converting the NDFSM       */
 4 s' = eps(s);
   /* The new starting-state will be the result of eps on the
      old start-state                                          */
   /* Compute δ':                                             */
 5 active-states = { s'};
 6 δ' = ∅;
 7 while ∃Q ∈ active-states s.t. δ' has not been computed yet for Q do
   |   /* Note that the elements of Q are sets of states!     */
 8 |   foreach Character c ∈ Σ_M do
 9 |   |   new-state = ∅;
10 |   |   foreach state q ∈ Q do
11 |   |   |   foreach state p s.t. (q,c,p)∈ Δ do
12 |   |   |   |   new-state = new-state ∪ eps(p);
13 |   |   |   end
14 |   |   |   Add the transition (Q,c,new-state) to δ';
   |   |   |   /* So we followed every possible transition for
   |   |   |      input c and got a set of states that we can reach
   |   |   |      for input c.  We now check if it is already in
   |   |   |      our active-states.  If it isn't we add this set
   |   |   |      of states as an element to active-states.   */
15 |   |   |   if new-state ∉ active-states then
16 |   |   |   |   active-states = active-states ∪ new-state;
17 |   |   |   end
18 |   |   end
19 |   end
20 end
21 K' = active-states;
22 A' = ∅ foreach Q ∈ K' do
   |   /* As there has to be only 1 accepting path, if the
   |      intersection between Q and A is not empty, we add Q to
   |      the accepting states                                 */
23 |   if Q ∩ A ≠ ∅ then
24 |   |   A' = A' ∪ Q;
25 |   end
26 end
```

Algorithm 3: ndfsmToDfsm

3.6 Theorems regarding FSMs

- **Theorem:** Every DFSM M, on input w ends in $|w|$ (length of w) steps.
- For each NDFSM there is an equivalent DFSM.

So by combining the above 2 statements we can conclude that the set of languages accepted by a DFSM is equal to the set of languages accepted by a NDFSM!

4 Lecture 4: Regular Expressions

A regular expression defines a regular language. Book chapters: 6.1 - 6.4.

4.1 What is Regex

- $\emptyset$ is a regular expression.
- ϵ is a regular expression.
- Every element of Σ is a regular expression.
- if α, β are regular expressions, then so is $\alpha\beta$.
- if α, β are regular expressions, then so i $\alpha \bigcup \beta$
- if α is a regular expression, then so is α^*.
- if α is a regular expression, then so is α^+
- if α is a regular expression, then so is (α)

4.2 Defining a language

Rules:

1. $L(\emptyset) = \emptyset$.
2. $L(\epsilon) = \{\epsilon\}$
3. $L(c) = \{c\}, foranyc \in \Sigma$.
4. $L(\alpha\beta) = L(\alpha)L(\beta)$.
5. $L(\alpha \bigcup \beta) = L(\alpha) \bigcup L(\beta)$.
6. $L(\alpha^*) = (L(\alpha))^*$
7. $L(\alpha^+) = L(\alpha\alpha^*)$ *note: if* $\alpha = \emptyset$ *then* $L(\alpha^+) = \emptyset$ *otherwise* α^+ *is formed by concatenating one or more strings drawn from* $L(\alpha)$
8. $L(\alpha) = L((\alpha))$

- 1, 3, 4, 5 and 6 give the language its power to define sets.
- Rule 8 is for grouping operators
- Rule 2 and 7 are solely there to clarify some of the functionality (We could do the same without these rules, but they do allow us to comprehend the language more easily).

4.3 Analysing regex

We can use the previous rules to analyse a regex expression. Meaning we can take a language and write it into a set form. We can also take the set form and write it into a language. The slides only provide examples.

4.4 Operator Precedence

1. Kleene star (exponentiation)
2. Concatenation (multiplication)
3. Union (addition)

This means that details matter:

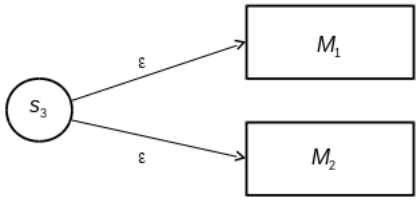
- $a^* \bigcup b^* \neq (a \bigcup b)^*$
- $(ab)^* \neq a^* b^*$

4.5 Regex to FSM Building blocks

In this section we will briefly describe some building blocks to "translate" regex to FSMs. Note that in the coming subsections we assume that the languages of β and γ are regular.

4.5.1 Union

Say we need to express $\beta \bigcup \gamma$ into a FSM. We can do this by introducing a new state with 2 epsilon transitions.



4.5.2 Concatenation

Now say we need to express $\beta\gamma$ (concatenation of β and γ) we need to do the following: For every accepting state in β add an epsilon transition to the starting state of Γ. (Yes it is that simple)

4.5.3 Kleene Star

Note that we also accept the statement if there is no input at all therefore we have to add a new starting state. This starting state is also an accepting state. From this starting state we can do an epsilon transition to the "old" starting state. After we have done this we add an epsilon transition from every accepting state to the old starting state. (Allowing the repetition of the pattern).

More formally:
Construct $M_2 = (\{s_2\} \bigcup K_1, \Sigma, \Delta_2, s_2, s_2 \bigcup A_1)$, where $\Delta_2 = \Delta_1 \bigcup \{((s_2, \epsilon), s_1)\} \bigcup \{((q, \epsilon), s_1) : q \in A_1\}$.

4.6 Theorems

- For every regular expression there is a corresponding FSM.
- **Kleene's Theorem**: FSMs and regular expressions define the same class of languages.

4.7 Algorithms

This section gives a nice overview of the various algorithms that were discussed during this lecture.

4.7.1 REGEX to FSM

Beginning with the primitive subexpressions of α and working outwards until an FSM for all of α has been built do:

Construct an FSM as described above. In other words we simply construct a Finite State Machine by using the building blocks described earlier. Note that you have to start with the primitives and work your way up.

4.7.2 FSM to REGEX heuristic

Data: Finite State Machine: M
Result: A Regular Expression equivalent to FSM M

```
1  Remove unreachable states from M;
2  if M has no accepting states then
3  |  return ∅;
4  end
5  if start state M part of a loop then
6  |  Create new Start State and connect to old start state with an ϵ
   |  transition;
7  end
8  if more than 1 accepting state in M OR transitions out of accepting state
   then
9  |  Create a new single accepting state;
10 |  Create an ϵ transition from all old accepting states to the new single
   |  accepting state;
11 |  All old accepting states now no longer accept;
12 end
13 if M has only 1 state then
14 |  return ϵ
15 end
16 while M contains more than start state and accepting state do
17 |  Select state r from M such that r is not accepting and is not the
   |  starting state;
18 |  Remove r from M;
19 |  Modify the transitions among remaining states s.t. M accepts the
   |  same strings.;
   |  /* This last step is hard.  You basically translate this
   |     state r into regex for the transitions in M. This
   |     leaves us finally with an actual Regular expression
   |     equivalent to M                                      */
20 end
21 return Regular Expression that labels remaining transition from start to
   accepting state;
```

Algorithm 4: FSM to REGEX heuristic

4.7.3 FSM to REGEX

```
   Data: Finite State Machine: M
   Result: A regular Expression equal to M
 1 Function fsmToRegex(FSM: M)
 2 |  M' = standardize(M: FSM);
 3 |  return buildregex(M');
 4 Function standardize(FSM: M)
 5 |  Remove unreachable states from M;
 6 |  if start state M part of a loop then
 7 |  |  Create new Start State and connect to old start state with an ϵ
   |  |  transition;
 8 |  end
 9 |  if more than 1 accepting state in M OR transitions out of accepting
   |  state then
10 |  |  Create a new single accepting state;
11 |  |  Create an ϵ transition from all old accepting states to the new
   |  |  single accepting state;
12 |  |  All old accepting states now no longer accept;
13 |  end
14 |  foreach state p,q ∈ M do
15 |  |  if more than 1 transition between p and q then
16 |  |  |  collapse the transitions into 1 transition;
17 |  |  end
18 |  end
19 |  if any transitions are missing then
20 |  |  Create these transitions;
21 |  end
22 Function buildregex(FSM: M)
23 |  if M has no accepting states then
24 |  |  return ∅;
25 |  end
26 |  if M has only 1 state then
27 |  |  return ϵ;
28 |  end
29 |  while M contains more than only start state and accepting states do
30 |  |  Select state r from M;
31 |  |  foreach transition in M do
32 |  |  |  p,q = states connected by transition;
33 |  |  |  if p != r && q != r then
   |  |  |  |  /* Compute a new label R' for transition p,q    */
34 |  |  |  |  R'(p,q) = R(p,q) ∪ R(p,r)R(r,r)* R(r,q);
35 |  |  |  end
36 |  |  end
37 |  |  Remove r and all transitions to it;
38 |  end
39 |  return regular expression that labels transition from start to
   |  accepting state
```

Algorithm 5: FSM to Regex

4.7.4 buildkeywordFSM

Say we want to match a pattern composed of a set of keywords, then we can use buildkeywordFSM. So basically we create a matcher that checks if a string contains any of the keywords!

Data: K: set of keywords
Result: An FSM accepting Strings that contain at least 1 entry of a given set of keywords

```
1 Create start state q_0;
2 foreach k ∈ K do
3 |  create a branch corresponding with k;
4 end
5 Create transitions for "dying" branches;
  /* "Dying" branches are branches that either end because the
     keyword has been found or there is a wrong continuation */
  /* If a keyword has been found loop on that last state
     regardless the input.  If there is a wrong continuation go
     back to the starting state.                              */
6 Make the states at the end of the branches accepting;
```

5 Lecture 5: Regular Grammar

5.1 Intro

Regular grammars are a third way of defining a regular language. A regular grammar G is a quadruple (V, Σ, R, S), where:

- **V** is the rule alphabet, which contains non-terminals and terminals
- Σ the set of terminals (subset of V)
- **R** is a finite set of rules of the form: $X \to Y$
- **S** is a non-terminal and is the start symbol.

5.2 Defining rules

all rules in R must:

- Have a left hand side that is a single non-terminal
- have a right hand side that is either:
  - ϵ
  - a single terminal
  - a single terminal followed by a single non-terminal.

The set of strings that a grammar G can generate is called L(G). We say that G *defines* the language L(G).

5.3 Theorems

Theorem: Class of languages that can be defined with regular grammars is exactly the regular languages.

5.4 Algorithms

5.4.1 Regular Grammar to FSM

Data: Regular Grammar G = (V, Σ, R, S)
Result: an FSM equivalent to G
Create a separate state for eacht nonterminal in V;
Start state is state corresponding to S;
if $\exists$ *a rule in R of form* $X \to w$ *s.t.* $w \in \Sigma$ **then**
  create new state labelled #;
end
foreach *rule of form* $X \to wY$ **do**
  add transition from X to Y labelled w;
end
foreach *rule of form* $X \to W$ **do**
  add transition from X to # labeled w;
end
foreach *rule of form* $X \to \epsilon$ **do**
  mark state X as accepting;
end
Mark state # as accepting;

Algorithm 6: Regular Grammar to FSM

6 Lecture 6: Non-Regular Languages

6.1 Regular or not?

Showing that a language is regular (easy) Showing that a language is not regular (hard)

6.1.1 Showing that a language is not regular

There are 6 ways to show that L is regular:

1. Show that L is finite.
2. Exhibit an FSM for L.
3. Exhibit a regular expression for L.
4. Show that the number of equivalence classes of $\approx_L$ is finite (The slides mention that we may forget this)

5. Exhibit a regular grammar for L.
6. Exploit the closure theorems (see theorems)

6.2 When use an FSM

FSMS perform best for repeating patterns. They aren't very efficient if the language is just a set of unrelated strings. However being regular does not necessarily mean that the language is tractable;

6.3 Closure Properties of Regular Languages

regular languages are closed under: (i.e. you should be able to apply these operations)

1. Union
2. Concatenation
3. Kleene star
4. Complement
5. Intersection
6. Difference
7. Reverse
8. Letter substitution

1-3 are proven by the constructions used in kleene's theorem. The rest can be proven as follows:

- For the complement we can swap the accepting and non-accepting states. (Note: Only works for deterministic and fully specified FSMs).
- Intersection: Can be constructed from previously proven operations: $L_1 \cap L_2 = \neg(\neg L_1 \cup \neg L_2)$.
- $L_1 - L_2$ same as intersection: $L_1 - L_2 = L_1 \cap \neg L_2$
- Start with an FSM for language L, first construct equivalent DFSM M. Then construct M' by
  - Inverting all directions of transitions in M.
  - Add new start state of and connect to all accepting states of M through ϵ transitions.
  - Make all accepting states non accepting.
  - Make original start state accepting.

Then M' accepts the language L^R

- Letter Substitution: Well we can simply use a function from Σ_1 to Σ_2

DON'T use closure **backwards**! (Closure holds in only 1 direction. e.g. If L_1 and L_2 are regular, then so is their intersection. But if we only know the intersection is regular then we may not conclude that $L_1 and L_2$ are regular. (Not even one of them).

6.4 Using The Pumping Theorem

Note: The theorem is given in the subsection Theorems of this section.

1. We don't know what value k has so we must state w in terms of k. (Honestly I have no idea what he means with this). So we choose a string w where —w— $\geq$ k.
2. Divide all possibilities for y into a set of equivalence classes that can be considered together.
3. Now for every class of possible y values where —xy— $\leq$ k and y $\neq \epsilon$ choose a value for q such that xy^qz is not in L.

6.5 Theorems

- **Theorem:** There is a countably infinite number of regular languages.
- **Theorem:** Every finite language is regular.
- **Pumping Theorem:** if M accepts any string of length —K— or greater, then that string will force M to visit some state more than once (thus traversing at least 1 loop). Or more formally:
  $\exists k \geq 1(\forall$ strings w $\in L, where |w| \geq k(\exists x, y, z(w = xyz, |xy| \leq k, y \neq \epsilon, and \forall q \geq 0(xy^qz \in L))))$ So basically this means that if we take a long string, there exists some part of that string that we can repeat any number of times and still be in the language that was defined. A "long" string is a string longer than the amount of states in the FSM.

7 Lecture 7: Context Free Grammars

Rewrite system: a list of rules and an algorithm for applying them. (aka **production system** or **rule-based system**

7.1 Grammars

- Expert systems
- Cognitive modelling

- Business practice modelling
- General models of computation
- **Grammars** (for defining languages)

7.1.1 Alphabets

A grammar is a set of rules that are stated in terms of 2 alphabets:

- **terminal alphabet**: Σ, contains the symbols that make up the string (in L(G))
- **non-terminal alphabet**: used for defining working symbols. These symbols will not (and may not) be present in the resulting string of the grammar. A grammar normally starts with the starting symbol S.

7.1.2 Deriving a string

When to stop?

1. When there are only terminal symbols left. (In this case working string has been **generated** by the grammar)
2. There are non-terminal symbols in the working string but none of them appears on the left-hand side of any rule in the grammar. Meaning we have a blocked or non-terminated derivation but no generated string.

It is possible that neither 1 nor 2 is achieved then we will end up in an endless string generation.

7.2 Regular grammars

Regular grammars produce strings one character at a time, moving left to right. However it may be more natural to describe generation more flexibly.

For example: L = ab^*a $S \to aBa$ $B \to \epsilon$ $B \to bB$
this brings us to Context-Free Grammars.

7.3 Context Free Grammars

Has no restrictions on the form of the right hand sides. Bur require single non-terminal on left handside.

So:
$S \to abDeFGab$
$S \to$
but not:
$ASB \to$

7.3.1 Examples

$A^n B^n$ $S \to \epsilon$
$S \to aSb$

Balanced parentheses $S \to \epsilon$
$S \to SS$
$S \to (S)$

7.3.2 Definition: Context-Free Grammars

A context-free grammar G is a quadruple (V, Σ, R, S) where:

- V is the rule alphabet (contains terminals and non-terminals)
- Σ = set of terminals, $\subseteq V$
- R = set of rules, finite subset of $(V - \Sigma) \times V^*$
- S = the start symbol, an element of $V - \Sigma$.

7.4 Derivations

if we can construct y from from x using grammar G in 1 step, we note this down as: $x \Rightarrow_G y$. the relation $x \Rightarrow_G y$ hold only iff $x = \alpha A \beta$ can transition to $y = \alpha\gamma\beta$ (this means that $A \to \gamma \in R$ must hold).

this allows us to say $w_0 \Rightarrow_G w_1 \Rightarrow_G w_2 \Rightarrow_G ... \Rightarrow_G w_n$. (this is a derivation in G). Let $\Rightarrow_G^*$ be the reflexive, transitive closure of $\Rightarrow_G$

we can then say that the language generated by G is:$\{w \in \Sigma^* : S \Rightarrow_G^* w\}$.

7.5 Recursive vs Self-Embedding Grammar Rules

A rule is **recursive** iff it is $X \to w_1 Y w_2$ where: $Y \Rightarrow^* w_3 X w_4$ for some w_1, w_2, w_3 and $w \in V^*$.

A grammar is recursive if it contains at least one self-embedding rule. We speak of a **self-embedding** rule iff it is: $X \to w_1 Y w_2$, where $Y \Rightarrow^* w_3 X w_4$ and both $w_1 w_3$ and $w_4 w_2$ are in Σ^+. Examples:

- $S \to aSa$ is self-embedding.
- $S \to aS$ is recursive but **not self-embedding**.

If a grammar G is not self-embedding then L(G) is regular. i.e. If a language L has a property that **every** grammar that defines it is self-embedding, then L is **not regular**

7.6 Designing Context-Free Grammars

- Generate related regions together:
  $A^n B^n$
- Generate concatenated regions: $A \rightarrow BC$
- SOmething

7.7 Concatenating independent languages

Let $L = \{a^n b^n c^m : n, m \geq 0\}$

8 Lecture 8: PDA: Push Down Automata

8.1 Definiton of a PDA

Recognizing Context-Free languages Two ways of recognizing:

1. Answer a yes or no question
2.

A PDA is in some sense a FSM BUT it has a **stack**.
$M = (K, \Sigma, \Gamma, Delta, s, A)$ where:

1. K is a finite set of states
2. Σ is the input alphabet
3. Γ is the stack alphabet
4. $s \in K$ is the initial state (starting state)
5. $A \subseteq K$ is the set of accepting states
6. Δ is the transition relation.

Δ is a finite subset of:

8.1.1 Configuration

Configuration of M = an element of $K \times \Sigma^* \times \Gamma^*$ and captures all the relevant information.

-

8.1.2 Manipulating the stack

will be written as cab
if $c_1, c_2, ..., c_n$ is pushed they will be added in order from right to left

8.1.3 Yields: $|-_M$ and $|-_M^*$

c is element of $\Sigma \bigcup \{\epsilon\}$ $\gamma_1, \gamma_2 and \gamma$ be any

8.1.4 Computation

Computation: finite sequence of configurations $C_0, C_1, ..., C_n$ for some $n \geq 0$ s.t.:

- C_0 = initial config
- C_n

8.1.5 Nonederminism

if M is in some configuration q_1, s, γ

8.1.6 Accepting

accepting computation: $C = (s, w, \epsilon)| -_M^* (q, \epsilon, \epsilon)$ and $q \in A$

- Process entire input string
- leave stack empty (at end)
- end in accepting state.

M **accepts** string w iff at least one of its computations accepts.
Language accepted by M: is set of all strings accepted by M.
A computation C of M is **rejected** iff:

- $C = (s, w, \epsilon)| -_M^* (q, w', \alpha)$
- C is not an accepting computation

8.2 Exploiting Nondeterminism

a PDA is **deterministic** iff:

- Δ contains no pairs of transitions that compete with each other
- whenever the machine is in an accepting configuration it has no available moves.

9 Lecture 9: LOL IDK

9.1 Proving Context-Free

9.2 Proving Not Context-Free

Can't use normal "Pumping" theorem. Instead we use pumping with parse trees. Perhaps better known as the **"Context-Free Pumping Theorem**.

9.2.1 Review of Parse Trees

Tree basics

- height: length of the longest path from root to any leaf
- branching factor: the number of splits of the node with the most splits.

Theorem: The length of the yield of any tree T with height h and branching factor b is $\leq b^h$. in other words:

if the string is "long" then its parse tree must look like: We choose the tree such that there is no other with fewer nodes.

9.2.2 What is k?

k serves 2 roles:

9.3 How long must w be?

if the height of the tree is bigger than n then some nonterminal occurs more than once

9.3.1 Regular vs Context-Free pumping theorem

- **Similarities:**
  - we choose w, the string to be pumped.
  - we choose a value for q that shows that w isn't pumpable.
  - We may apply closure theorems before we start.
- **Differences:**
  - in regular case: 1 part of string is being pumped, CF: two regions, v and y must be pumped in tandem.
  - CF: We don't know anything about where in the strings v and y will fall. all we know is that they are reasonably "close together" i.e. —vxy— $\leq$ k.

10 Lecture 10: Turing Machines

Turing machines have 2 properties:

1. Powerful enough to describe all computable tings
2. Simple enough to ..

Instead of a stack it uses an infinite tape (elongates in 2 directions). each step consists of 3 actions: Choose next state, write current square and move left or right.

10.0.2 Formal definition

Turing machine M is a sixtuple: $(K, \Sigma, \tau, \delta, s, H)$

- K is a finite set of states;
- Σ is the input alphabet not containing $\square$.
- τ is the tape alphabet, which must contain $\square$ and has Σ as a subset.
- $s \in K$ is the initial state.
- $H \subseteq K$ is the set of halting states.
- δ is

Notes:

- δ is a function not a relation
-

Transitions are labelled with x/t/a, meaning that a transition is taken if the current character is an x, in which case a t is written and the head moves according to action a. We omit unusable transitions.

10.1 Programming Turing Machines

10.1.1 Halting

An FSM is guaranteed to halt. However a PDA is not not guaranteed to always halt. However A PDA can be rewritten in such a way that it is guaranteed to halt. For Turing machines however this is not the case, there is no guarantee that it will halt.

10.2 Formalizing the Operation

Yields: $(q_1, w_1) \vdash_M (q_2, w_2) iff (w_2, w_2)$ is derivable, via δ in one step. Configuration C_1 **yields**

10.3 Notations for Turing Machines

In this section some basic machines are described; s denotes the starting state, h a halting state.

- **Symbol writing machine**: For each $x \in \tau$ define M_x written just , to be a machine that writes x and then halts (formally it must move then left or right, but will move back).
- **Head moving machines**: R: for each $x \in \tau, \delta(s, x) = (h, x, \rightarrow)$
  L: for each $x \in \tau$

- **Checking inputs** first run some machine M_1 until it halts if the condition is true, run M_2
- **Combining Machines:**

10.3.1 Short hands

Short hands: Variables: 2 ways of implementation:

- Write the value for the variable somewhere on tape (req. lots of scanning)
- Use separate states for all possible values of a variable (req. big machine)

Regardless of the machine we hide the details: **Using Variables**

- in a condition:
- write the value of a variable: x

10.3.2 Useful Machines

- Find the first occurrence of 'a' to the left of the current square.
- Find the first occurrence of 'a' or 'b' to the right of..

10.4

Language recognition:

11 Lecture 11: Turing Machine Extensions & Unrestricted Grammars

Many extensions can be added to turing machines. However: **For every extended machine there is an equivalent basic machine**.

By converting an extended machine into a basic one we can set a bound on the complexity of the machine.

11.1 Various Extensions

11.1.1 Multiple Tapes

We can use more than 1 tape, giving it more than 1 head location. This requires a different transition function: instead of

- ((K-H) x Γ_1
- ((K-H) x Γ_2
- etc.

we need something like

- K x Γ_1 x { $\leftarrow, \rightarrow, \uparrow$}
- K x Γ_2 x { $\leftarrow, \rightarrow, \uparrow$}
- K x Γ_3 x { $\leftarrow, \rightarrow, \uparrow$}
- etc.

This means that we can switch between heads by going up.
However adding a tape adds not power. Let M be a k-tape Turing machine..

11.2 Impact of adding nondeterminism

- FSMs
  - **Power?** NO
  - **Complexity**
    - **Time?** NO
    - **Space** Yes (increase)
-

11.2.1 Definition Nondeterministic TM

A nonderterministic Turing Machine is a sixtuple $(K, \Sigma, \Gamma, \Delta, s, H)$

- Δ is

11.2.2 Deciding and SemiDeciding

Deciding: a Non Deterministic Turing Machine accepts a string iff at least one of its computations accepts. It rejects if all of its computations reject.

A machine decides a language L $\subseteq \Sigma^*$ iff, $\forall w$:

-

Semideciding:

11.3 Nondeterministic Function Computation

What do when multiple paths have multiple results? or when some paths halt and some not?

In this case we use a **Strict** definition: a nondeterministic machine computes a function f iff, $\forall w \in \Sigma^*$

Theorem: If a nondeterministic TM M decides or semidecides a language, or computes a function, then there is a standard TM M' semideciding or deciding the same language or computing the same function.

11.4 Simulating a real computer

Theorem: A random-access, stored program computer can be simulated by a turing machine. if the computer requires n steps to perform some operation, the turing machine simulation will require $O(n^6)$ step. in order to simulate a computer we will use 7 tapes:

- Tape 1: the computer's memory
- Tape 2: The program counter.
- Tape 3: The address register

11.5 Context-free Grammars

An **unrestricted grammar** G is a quadruple (V, Σ, R, S) where:

- V is an alphabet
- $\Sigma$the set of terminals ($\subseteq V$)
- R is the set of rules and is a finite subset of $(V^+ \times V^*)$.
- S is the starting symbol (element of V - Σ)

One of the powerful aspects of context-free grammars is the ability to add more than 1 non-terminal when transitioning. Another powerful property would be the ability to require a specific character to be in front or after a non-terminal. e.g.:

- $S \to aBSc$
- $Ba \to aB$ (basically moves the a to the left of B)

Context-free grammars have a strong procedural feel. Overall we have 3 main phases:

- Generate the right number of various symbols.
- Move them around to get them in the right order.
- Clean up and get rid of non-terminals.

Theorem: A language is generated by an unrestricted grammar if and only if it is in SD (semi-decidable languages).

12 Lecture 12: Intro Analysis of Complexity

So, when is a language "easy to decide"?

Our theory on complexity only applies to **decidable** languages. This is because we need deciding Turing Machines. We will only look at decision problems, this decreases the complexity of the output and thus the length of the answer does not affect the comlexity of our machine.

12.1 Encoding Types

- **Integers**: just don't use base 1.
- **Graphs:** Use an adjacency matrix Alternatively use a list of all edges.

12.2 Choosing a Model of Computation

We will use Turing machines in order to provide a model of computational complexity. We will use 1 tape of fixed size. It does not matter whether the TM is deterministic of nondeterministic. A TM differs at most a sixth power of the number of steps for a "real" computer.

Now we define a function: timereq(M) as a function of n:

- if M is a **deterministic** TM that halts on all inputs then:
  spacereq(M) = f(n) = the maximum number of tape squares that M reads on any input of length n.
- if M is a **nondeterministic** TM all of whose computational paths halt on all inputs, then:
  timereq(M) =

12.3 Asymptotic Dominance

$\mathcal{O}$

We say that $f(n) \in \mathcal{O}(g(n))$ iff

12.3.1 Using $\mathcal{O}$

Suppose timereq(M) $= 3n^2 + 23n + 100$, then:
timereq(M) $\in \mathcal{O}(n^2)$

So in short: $\mathcal{O}(c) \subseteq$

12.3.2 Asymptotic strong upper bound

$f(n) \in \mathcal{O}(g(n))$ iff, for every positive c, there exists a positive integer k and a positive constant c such that: $\forall n \geq k(f(n) \geq cg(n))$ (so it is never touching)

12.4 Algorithmic Gaps

We would like to get an upper bound (there exists an algorithm that decides L and that has complexity C_1 We also would like to have a Lower bound (Any algorithm that decides L must have complexity at least C_2 And ultimately we want these bounds to be equal $C_1 = C_2$ (meaning we have the optimal algorithm). However this is often hard to prove.

13 Lecture 13: Time Complexity Classes

13.1 Complement

13.2 The Language Class P

Which lectures are in P?

- Every Regular language
- Every context-free language since there exist context-free parsing algorithms that run in $\mathcal{O}(n^3)$ time.
- Others:
  - $a^n b^n c^n$

  how to show this?
- Describe a one-tape, deterministic Turing machine.
- May use multiple tapes; however this comes at a price: **Polynomial Time**.
- State an algorithm that runs on a conventional computer. Price: **polynomial**.

13.3 Graph Languages

13.3.1 Definitions:

- **Vertex Degree**: # of edges with it as an endpoint
- **Sub-Graph Isomorphism**

13.3.2 Theorems

- **Euler's Path** Graph connected and exactly 2 vertices with odd degrees? -¿ There exists an eulerian path with the 2 odd degree vertices as end and start point.
- **Ealer's Circuit** Graph connected and all vertices have even degree? -¿ There exists an eulerian circuit.

13.4 The Language Class NP

$L \in NP$ iff:

- There is some NDTM

alternative definition: L $\in NP$ iff three exists a deterministic TM V such that: These definitions are in fact equivalent to each other.

13.4.1 Deterministic Verifying

13.4.2 3-SAT

14 Lecture 14: NP-Complete

14.1 NP-Completeness

If a language is

1. L is in NP
2. Every language in NP is deterministic, polynomial-time reducible to L.

L is **NP-hard** iff it possesses property 2. L is **NP-complete** iff it has both properties. All NP-Complete languages can be viewed as being equivalently hard.

14.1.1 Showing L is NP-Complete

We need a NP-Complete language L' : The idea is to find a transition from all NP languages to L'

So the problem that we face first is finding the **first** NP-Complete language.

14.1.2 Cook-Levin Theorem

Define: SAT = { w:w is awff in Boolean logic and w is satisfiable} Theorem: SAT is NP-Complete. We already Cook-Levin **does not have** to be learned.

14.2 Proving that L is NP-Complete

Theorem: if L_1 is NP-Complete, $L_1 \leq_p L_2$ and L_2 is in NP, then L_2 is also NP-Complete.