Theory of Computation

Summary

Reg Dodds
Department of Computer Science
University of the Western Cape

rdodds@uwc.ac.za

CSC 311 Paper I: Theory of Computation Paper and Supplementary

Friday 20th January 2017 at 14h30

Venue: Proteaville Recreation Hall
Located at the corner of:
Abdurahman Street and Peter Barlow Drive

Bellville South

Note:
I do not take any responsibility for informing you of venue or timetable changes.

Euclid (circa 300 BCE) wrestled with what geometrical figures could be constructed with a *compass and a ruler*.

John Napier (1550–1617) by 1614, had already published his work on *logarithms*, making hand calculations very efficient.

Blaise Pascal (1623–1662) built an *additive calculator* in 1642.

During 1670 Isaac Barrow (1630–1677) published his "Lectiones Geometricae", which introduced many rules of differential calculus before Newton or Leibniz.

Gottfried Leibniz (1646–1716) improved the Pascaline in 1694–1703 so that it could multiply. Leibniz also dreamed of a more general logic machine.

Georg Cantor (1846–1918) in 1891 invented diagonalization and that enabled Gödel's discovery.

Soon after in 1900 David Hilbert (1862–1943) said, "*We must know! We will know!*," He stated that all of mathematics could, and ultimately would, be built on unshakable logical foundations. Hilbert's great rallying cry was "*in mathematics there is no ignorabimus.*"

Kurt Gödel (1906–1978) in 1931 published an astonishing discovery that proved Hilbert wrong. There are statements in arithmetic that cannot be *known*.

This was followed by a flurry of work that showed how to do computation and what was impossible using that notation.

Alonzo Church (1903–1995), using the *lambda calculus*, showed that the decision problem was undecidable in 1936.

Alan Turing (1912–1954), using a *Turing machine*, showed that the decision problem was undecidable in 1936.

Stephen Kleene (1909–1994) established *general recursive functions* as a basis for computation in 1936.

Emil Post (1897–1954) in 1943 published work on his *rewriting machines*.

John von Neumann (1903–1957) showed in 1946 how to *build a computer* which in which data and instructions shared the same memory.

In this course we will show what it is *that can and cannot be computed*.

Introduction to the course

We will introduce a hierarchy of machines with a finite number of components.

The simplest machines have *no memory*. These are the finite state automata, that are either deterministic or non-deterministic. These machines can be formulated as *regular grammars* or as *regular expressions*.

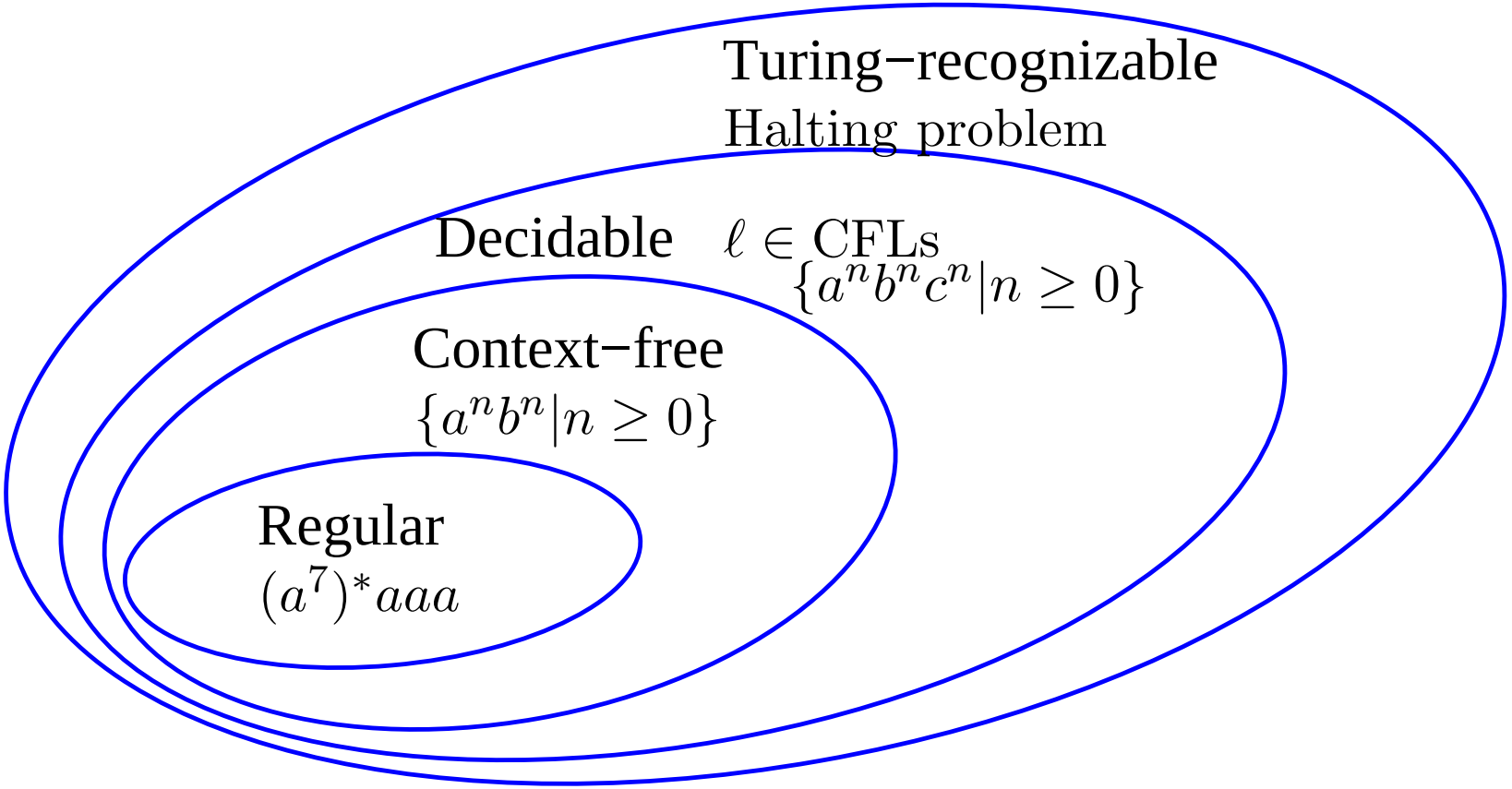
A slightly more complex machine has a *single memory stack*. These are nondeterministic pushdown automata that can also be written as *context-free grammars*.[1]

Machines with *two stacks* are shown to be as powerful as machines with *random access memory*. These automata can also be written as *Turing machines* (TMs). TMs are finite state machines with an unbounded tape and a read-write head that can read or write on the tape and move in either direction.

More powerful machines do not exist.

[1] Stacks or memory have no bounds on their size.

The Hierarchy of Machines and Languages



The introductory chapter covers important concepts that are assumed in this course.

- Automata, computability and complexity
- Mathematical notions and terminology
  sets, sequences and tuples, functions and relations, graphs, strings and languages, boolean logic
  proofs by
  —construction
  —contradiction
  —induction
- Powerset—prove that if A is a set then $|\mathscr{P}(A)| = 2^{|A|}$

Regular languages

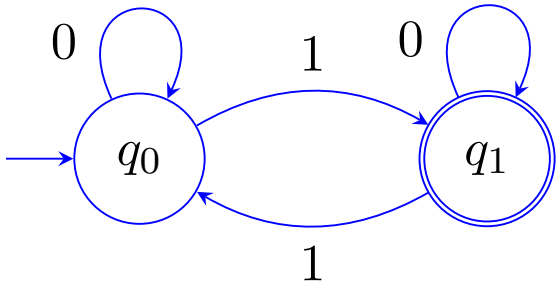
- A Deterministic finite automaton (DFA) as a Grammar
- Deterministic finite automata (DFAs)
- Equivalent representations of DFAs

  1. Graphically as a *state-transition diagram*
  2. Grammatically as a *regular grammar*
  3. Algebraically as a *regular expression*
  4. Functionally as a *transition table*
  5. Programmed

     (a) *with states as callable procedures*
     (b) *as an array driven process*

- Definition of a DFA, nondeterminism, regular languages the set of all regular languages $\mathscr{L}_r$, pumping lemma for regular languages, languages that are not regular languages

Implementing a DFA for the odd-parity problem



δ	0	1
$\rightarrow q_0$	q_0	q_1
$*q_1$	q_1	q_0

The odd-parity DFA using states as callable procedures.

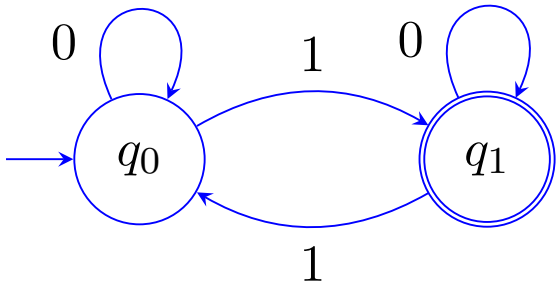
```
def q0(w):
  if len(w) == 0:
    return "reject"
  elif w[0] == '0':
    return q0(w[1:])
  elif w[0] == '1':
    return q1(w[1:])
  else:
    return "reject"
```

```
def q1(w):
  if len(w) == 0:
    return "accept"
  elif w[0] == '0':
    return q1(w[1:])
  elif w[0] == '1':
    return q0(w[1:])
  else:
    return "reject"

w = '1110000011'
print(w, q0(w))
```

Implementing a DFA for the odd-parity problem

The odd-parity DFA using an array driven loop.



δ	0	1
$\rightarrow q_0$	q_0	q_1
$*q_1$	q_1	q_0

```
1  start = 0
2  F = [False, True]
3  Q = [[0,1], [1,0]]
4
5  state = start
6  w = '11101000011'
7  while len(w) > 0:
8     letter = int(w[0])
9     state = Q[state][letter]
10    w = w[1:]
11
12 if F[state]:
13    print ("accept")
14 else:
15    print ("reject")
```

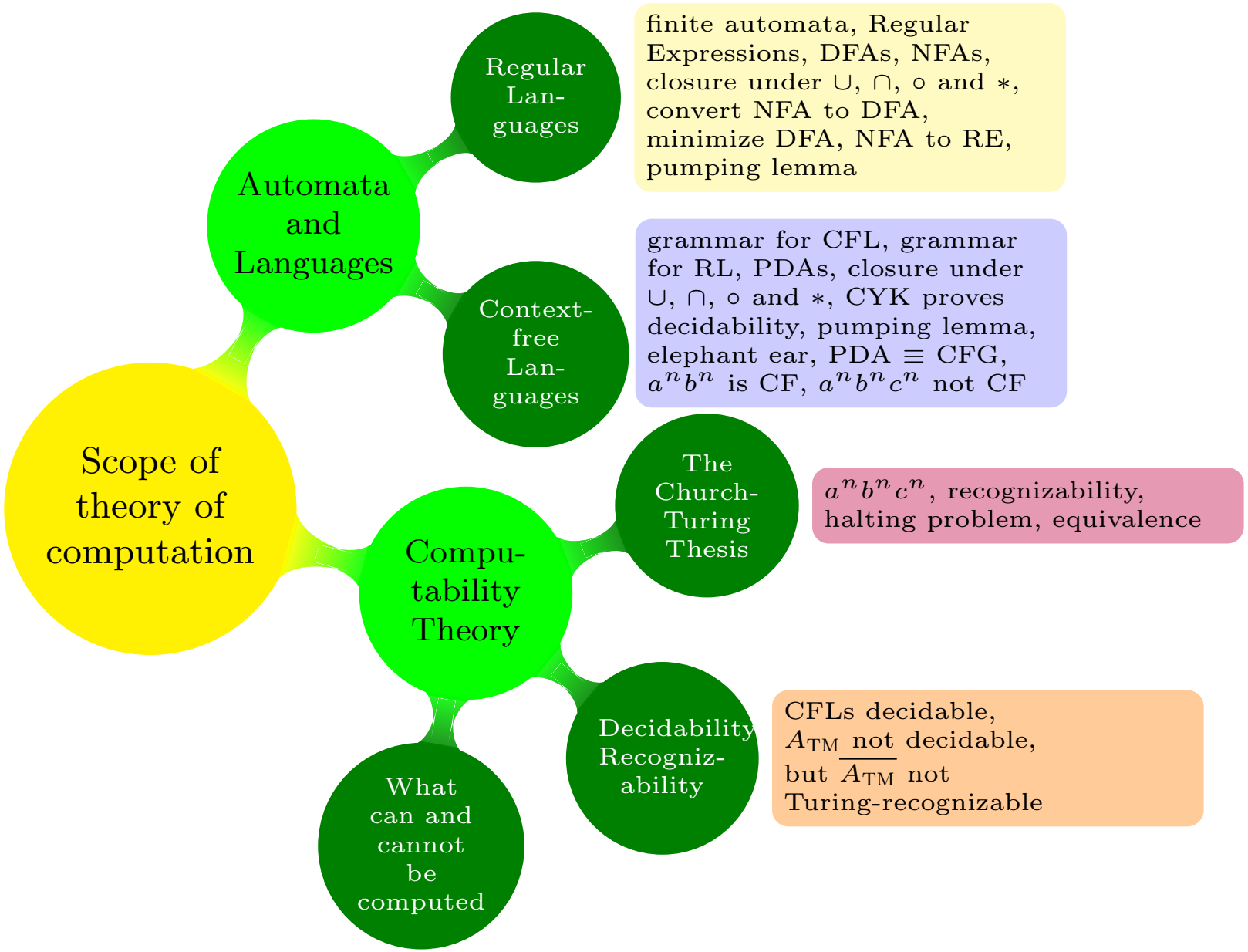
Memo for 2016 ToC examination—Introduction

- The scope of the examination is all of the first four chapters of Sipser.
- Do all the exercises in those four chapters.
- Know the flow of ideas through

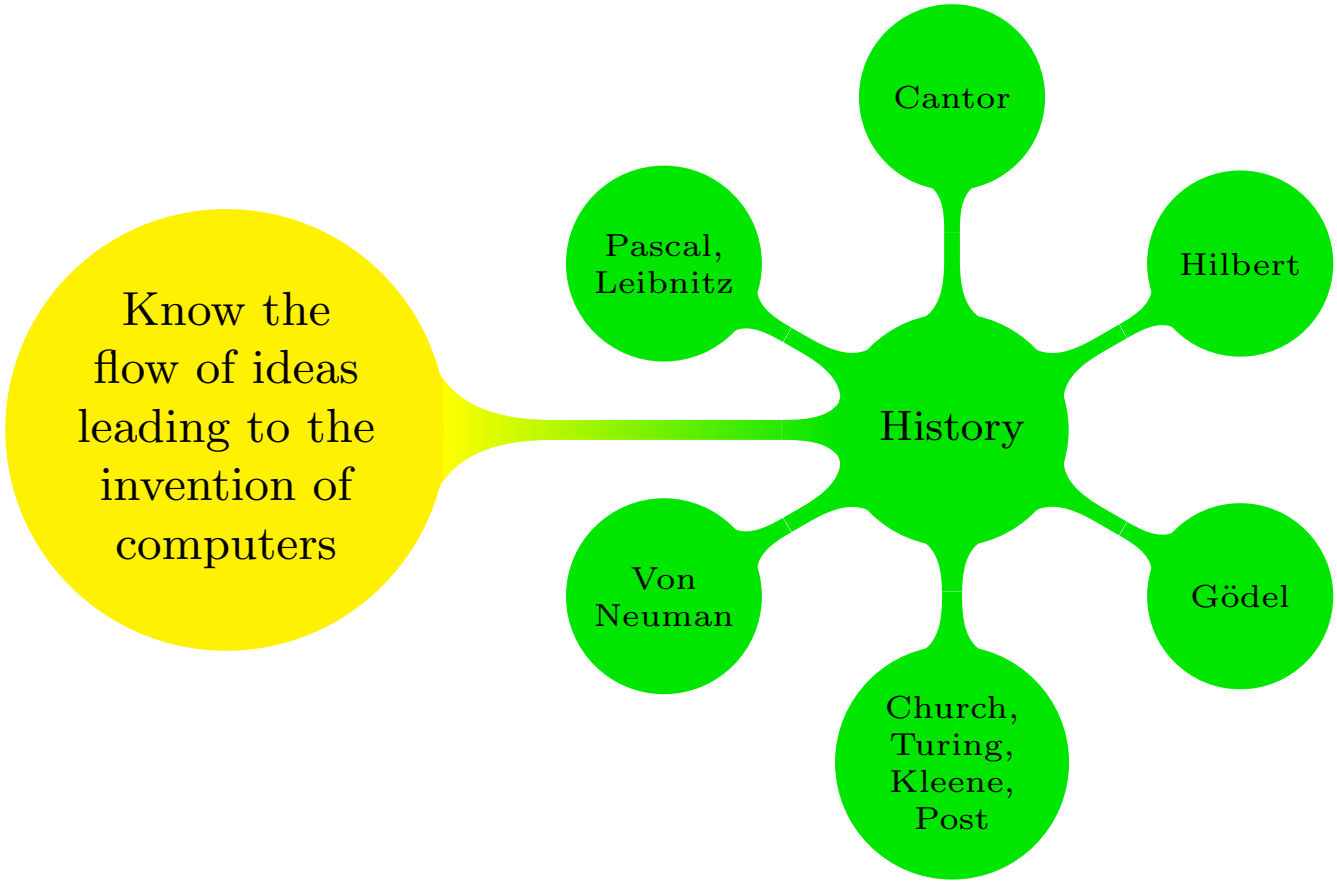
  Cantor—Hilbert—Gödel—Church, Turing, Post and Kleene—Von Neuman

  for the invention of digital computers.
- Topics such as minimization of DFAs, and the CYK algorithm are included although they appear elsewhere as problems in Sipser.
- These notes intend to be a check list of the work covered.

Memo for 2016 ToC examination



Memo for 2016 ToC examination—Introduction



Memo for CSC 311 examination 2016—DFAs, Regular expressions

- Define a deterministic finite automaton (DFA), pronounced *dee-eff-ay*, so we write "a DFA" and "many DFAs".
- Represent finite automata in a variety of ways, using a state transition table, a state chart or diagram, a regular grammar or a regular expression.
- Implement finite automata with each state as a procedure call.
- More powerful is the general implementation of finite automata as a state transition table of next states.

Memo for CSC 311 examination 2016—DFAs, Regular expressions

- Get the union of two DFAs N_1 and N_2 by combining their transition functions.
- OR get union of two DFAs by construction of a non-deterministic finite state automaton (NFA, pronounced *enn-eff-ay*—so we write "an NFA" and "many NFAs").
- Build the concatenation of these two DFAs $N_1 \circ N_2$ with a constructed NFA.
- Given a DFA N, build N^* with NFA construction.
- Define a regular expression.

Memo for CSC 311 examination 2016—DFAs, Regular expressions

- Convert regular expression to NFA—do easy ones using common sense.
- If converting regular expression to NFA by common sense becomes overwhelming—do the conversion element-by-element using the definition of a regular expression and the equivalence of each construct in the definition with its corresponding NFA.
- Convert NFA to DFA—using subset construction.
- What is the time complexity to convert an NFA to a DFA using subset construction?
- Minimize a DFA.
- Convert DFA to regular expression using generalized NFA by ripping.
- Show that regular expressions are equivalent to finite automata.

CSC 311 2016 memo—Pumping lemma

- Give the pumping lemma for regular languages.
- Prove the pumping lemma for regular languages.
- The pumping lemma is not used for showing regularity. For what is the pumping lemma useful?
- There are some examples on the next slide

- Show with or without the pumping lemma for regular languages that:

1. $\{a^n | n \geq 0 \text{ and } a \in \Sigma\}$ is regular.
2. $\{a^n b^n | n \geq 0 \text{ and } a, b \in \Sigma\}$ is not regular.
3. $\{a^2 b^2 | a, b \in \Sigma\}$ is regular.
4. $\mathscr{R} = \{a^j b^k c^\ell | i \geq 0, j \geq 0 \text{ and } \ell \geq 0, \text{ and } a, b, c \in \Sigma\}$ is regular.
5. $\mathscr{T} = \{a^n b^n c^n | n \geq 0 \text{ and } a, b, c \in \Sigma\}$ is not regular.
6. $\mathscr{T} \subseteq \mathscr{R}$.
7. $\{a^3 b^3 c^3 | a, b, c \in \Sigma\}$ is regular.
8. $\{a^{2n} | n = 3 \text{ and } a \in \Sigma\}$ is regular.
9. $\{a^{2n} | n \geq 0 \text{ and } a \in \Sigma\}$ is not regular.
10. $\{a^{2^n} | n = 3 \text{ and } a \in \Sigma\}$ is regular.
11. $\{a^{2^n} | n \geq 0 \text{ and } a \in \Sigma\}$ is not regular.
12. $\{a^n | n \in \{\text{Fibonacci numbers}\} \text{ and } a \in \Sigma\}$ is not regular.
13. $\{ww^{\mathcal{R}} | w \in \Sigma^*\}$ is not regular.
14. $\{wxw^{\mathcal{R}} | w \in \Sigma^* \text{ and } x \in \Sigma\}$ is not regular.
15. $\{ww | w \in \Sigma^*\}$ is not regular.

CSC 311 2016 memo—Regular expressions in languages

- Most programming languages have regular expression engines for manipulating substrings in text.
- These regular expressions are written using simple rules. What do the following regular expressions (REs) mean?

1. `[0-9]`, `[0-9]*`, `[0-9]+`, `[0-9]{3}` and `[0-9]{1-8}`
2. Write `[0-9]` and `[a-z]` in another way.
3. `[a-zA-Z]`, `[a-zA-Z]*`, `[a-zA-Z]+` and `[^a-zA-Z]`
4. `[^*/]`, `"/*"`, `"*/"`, `[*/]`, `"if"`, `"else"`
5. `[.]`, `[^.]`, `[.]*`, `[.]+`, `.*`, `[.*]*`, `[.*]`, `[*]*`
6. `[+-]?[0-9][.][0-9]{0-7}([Ee][+-]?[0-9]{1-3})?`
7. `[+-][0-9]?[.][0-9]{0-8}[Ee]?[+-][0-9]{1-3})`
8. Give an RE for a Java variable name.
9. Give an RE to match `the`, `a`, `an` in a text. It need not match these words at the start of a sentence.
10. Adapt the previous to match these words anywhere in a sentence.
11. Give an RE to match palindromes—for experts only.

Let $A = L(G)$, $G = (V, \Sigma, R, S)$ is a context-free language, and $w = \Sigma^*$.

- Grammar. CF grammars can be used to define a language.
- What characterizes a regular grammar?
- When is a grammar context-free?
- Give an example of a grammar that is neither context-free or regular.
- Given a context-free language, give its grammar, such as for: palindromes over Σ; $\{a^n b^n | n \geq 0$ and $a, b \in \{a, \}\}$; a grammar for balanced parentheses; unambiguous grammar for arithmetic over the variables `a`, `b` and `c`; etc.
- What is a parse tree? When is a CFL ambiguous?
- Using the Cocke-Younger-Kasami procedure determine if $w \in A$.
- If $w \in A$, give its right-most and left-most derivation $S \stackrel{*}{\Rightarrow} w$ in each case.
- What are the productions of a grammar in Chomsky normal form (CNF).
- Given G, convert it into Chomsky normal form.
- If a grammar has the productions $S' \to S$; $S \to SaSbScSdSeSfSgSiShS$, $S \to \varepsilon$, how many productions need to be *added* to it to bring it to Chomsky normal form.
- See Sipser's Example 2.10 on Page 110 of 2nd Ed. to convert rules to CNF.

CSC 311 2016 memo—Non-deterministic pushdown automata (PDA)

- What is a PDA?—Remember PDA stands for a non-deterministic pushdown automaton. Deterministic PDAs (DPDAs) are a bit weaker than PDAs but are easier to code. Programming languages usually need only a DPDA.
- Explain how to construct a DFA for $\{a^n b^n | n = 10^{100} \text{ and } a, b \in \Sigma\}$. How many states will it require? Can a program that simulates such a DFA be implemented in 2016?
- PDA is a 6-tuple $(Q, \Sigma, \Gamma, \delta, q_0, F)$.
- PDA is more powerful than DFA because it has a memory stack. It can count.
- PDA can be represented by a state diagram or transition table, and can be programmed.
  Give a PDA and CF grammar for:

  1. $\{a^n b^n | n \geq 0 \text{ and } a, b \in \Sigma\}$.
  2. $\{a^j b^k c^\ell | j, k, \ell \geq 0, \text{ and } j = k \text{ or } j = \ell \text{ and } a, b, c \in \Sigma\}$.
  3. The even palindromes $\{ww^{\mathcal{R}} | n \geq 0 \text{ and } a, b \in \Sigma\}$.
  4. The odd palindromes.

CSC 311 2016 memo—Non-deterministic pushdown automata (PDA)

- PDAs can be used to define languages.
- A language is context free *if and only if* some PDA recognizes it.
  This is proved in two directions. The one direction uses the elephant's ear to construct a PDA from a grammar and the other direction shows how to construct rules for a grammar from the transitions of a PDA.
- The transition from state p to state q in a PDA with the input a, that pops y off the PDA's stack and writes z in its place, is written as $a, y \to z$.
- $a, S \to vwx$ is shorthand notation
  for the three transitions from state p, via q and r, to state s,
  (1) First from p to q with $a, S \to x$, next from
  (2) q to r with $\varepsilon, \varepsilon \to w$ and finally from
  (3) r to state s with $\varepsilon, \varepsilon \to v$.
  This implements the transition $\delta(p, a, S) \ni (s, vwq)$.
- Since every regular language has a DFA that recognizes it, and every DFA is a PDA that ignores its stack, it means that every regular language is also context-free language.

CSC 311 2016 memo—Pumping lemma for context-free languages

- The pumping lemma for context-free languages is used to show that a language is not context-free.
- Take a non-empty string that is long enough from the language L and show that it cannot be pumped then L is not context free.
- Sketch the proof.

CSC 311 2016 memo—Pumping lemma

Give grammar and PDA when it is context-free, or show with the pumping lemma for context free languages that it is not:

1. $\{a^n | n \geq 0 \text{ and } a \in \Sigma\}$ is context-free.
2. $\{a^n b^n | n \geq 0 \text{ and } a, b \in \Sigma\}$ is context-free.
3. $\{a^2 b^2 | a, b \in \Sigma\}$ is context-free.
4. $\mathscr{R} = \{a^j b^k c^\ell | i \geq 0, j \geq 0 \text{ and } \ell \geq 0, \text{ and } a, b, c \in \Sigma\}$ is context-free.
5. $\mathscr{T} = \{a^n b^n c^n | n \geq 0 \text{ and } a, b, c \in \Sigma\}$ is not context-free.
6. $\mathscr{T} \subseteq \mathscr{R}$.
7. $\{a^3 b^3 c^3 | a, b, c \in \Sigma\}$ is context-free.
8. $\{a^{2n} | n = 3 \text{ and } a \in \Sigma\}$ is context-free.
9. $\{a^{2n} | n \geq 0 \text{ and } a \in \Sigma\}$ is context-free.
10. $\{a^{2^n} | n = 3 \text{ and } a \in \Sigma\}$ is context-free.
11. $\{a^{2^n} | n \geq 0 \text{ and } a \in \Sigma\}$ is / is not context-free.
12. $\{a^n | n \in \{\text{Fibonacci numbers}\} \text{ and } a \in \Sigma\}$ is / is not context-free.
13. $\{ww^{\mathcal{R}} | w \in \Sigma^*\}$ is context-free.
14. $\{wxw^{\mathcal{R}} | w \in \Sigma^* \text{ and } x \in \Sigma\}$ is context-free.
15. $\{ww | w \in \Sigma^*\}$ is / is not context-free.

CSC 311 2016 memo—Prove with grammar or PDA or use pumping lemma

Give grammar and PDA when it is context-free, or show with the pumping lemma for context free languages that it is not:

1. $\{a^n b^n a^n b^n | n \geq 0 \text{ and } a, b \in \Sigma\}$ is not context-free.
2. $\{a^n b^n c^n d^n | n \geq 0 \text{ and } a, b, c, d \in \Sigma\}$ is not context-free.
3. $\{a^n b^n c^m d^m | n \geq 0 \text{ and } a, b, c, d \in \Sigma\}$ is context-free.
4. $\{a^n b a^{2n} b a^{3n} | n \geq 0, a, b \in \Sigma\}$ is not context-free.
5. $\{www | w \in \Sigma^*\}$ is / is not context-free.

- A 2PDA has two stacks is all powerful.
- All the problems above that are not context-free can be implemented with 2PDAs.
- Try to create 2PDAs for them.

CSC 311 2016 memo—Turing machines

- Turing machine (TM) is more powerfully that a single stack PDA because tape can be read or written. Most powerful form of automaton.
- Show sequence of configurations given a TM.
- Adding memory tapes does not increase power.
- Making TM non-deterministic does not increase power.
- What is an algorithm.
- Church-Turing thesis.
- Hilbert's 10th problem.

CSC 311 2016 memo—Decidability

- The language A_{DFA} is decidable because we can build a machine that will either accept or reject a string.
- The language A_{NFA} is decidable because we can build a machine that will either accept or reject a string.
- The language A_{REX} is decidable because we can build a machine that will either accept or reject a string.
- The language $E_{\text{DFA}} = \{\langle A \rangle | A \text{ is a DFA and } L(A) =\}$ is decidable. Why?
- The language $EQ_{\text{DFA}} = \{\langle A, B \rangle | A \text{ and } B \text{ are DFAs and} L(A) = L(B)\}$ is decidable. Understand and be able to explain.
- The language A_{CFG} is decidable because we can build a machine using CYK to decide if any string has a derivation or not.
- The language $E_{\text{CFG}} = \{\langle A \rangle | A \text{ is a CFG and } L(A) =\}$ is decidable. Why?
- The language $EQ_{\text{DFA}} = \{\langle A, B \rangle | A \text{ and } B \text{ are DFAs and} L(A) = L(B)\}$ is not decidable. Understand and be able to explain.
- Every CFG is decidable.

- The language $A_{\text{TM}} = \{\langle M, w\rangle | M \text{ is a TM and } M \text{ accepts } w\}$ is Turing-recognizable.
- The language $A_{\text{TM}} = \{\langle M, w\rangle | M \text{ is a TM and } M \text{ accepts } w\}$ is undecidable.
- If a set maps one-to-one onto $\mathbb{N}$ it is countable or enumerable.
- Since the mapping $J[p, q] \equiv (p+q-1)(p+q-2)/2+q$ maps every pair of in $\mathbb{Q}$ onto a unique number in $\mathbb{N}$. The set of rational numbers $\mathbb{Q}$ is countable.
- The set of real numbers $\mathbb{R}$ is undenumerable of uncountable and it is proved with Cantor's diagonalization method.
- There exist languages that are not Turing-recognizable.
- A_{TM} is undecidable.
- The language $\overline{A_{\text{TM}}}$ is not Turing-recognizable.

Memo for CSC 311 examination 2016—Compatibility and NP complexity

This is to remind you of important background material from CSC212 that you should have mastered on or before 2015.

- Be able to prove that the halting problem is non-computable.
- Define totality, equivalence.
- What is meant by partial computability?
- What are proof systems?
- Give a class of algorithms that are considered efficient?
- When do we consider an algorithm intractable?
- What is meant by **NP** completeness?
- Understand and be able to explain what the practical implications are of knowing that an algorithm is **NP** complete.
- Give a list of at least 5 problems that are **NP** complete.
- Be able to explain how to go about proving that a given problem is **NP** complete.

Memo for CSC 311 examination 2016—Layout of paper

Similar layout as last year's paper.

That's all folks

That's all folks!