CS 3100 – Models of Computation – Fall 2011 – Notes for L22

Contrapositives Fact: Suppose we have a theorem of the form $A \Rightarrow B$. Then we have also a contrapositive theorem: $\neg B \Rightarrow \neg A$.

A Useful Theorem about Regular Sets

Theorem "Immiscible Union": Suppose $A \cap B = \emptyset$ and A is non-regular, and B is regular (there is a DFA for it called DFA_B). Then $A \cup B$ is non-regular.

Proof: Suppose not. Then there is a DFA for $A \cup B$—call it $D_{A \cup B}$. Build a DFA for A by taking $D_{A \cup B} \cap ((\overline{DFA_B}))$. This is a DFA for A (call it D_A) because it accepts exactly when $D_{A \cup B}$ accepts and DFA_B rejects – so must be a string exactly within A (since there is no overlap between A and B). This is a contradiction because A is non-regular, so it can't have a DFA.

Pumping Lemma

The regular pumping lemma we studied is of the form $Regular(L) \Rightarrow RHS$ where

$$
\begin{array}{l}
RHS = \\
\quad \exists n \in N : \\
\quad \forall w \in L \ : |w| \geq n \\
\qquad\qquad \Rightarrow \\
\qquad\qquad \exists x, y, z \in \Sigma^* \ : \\
\qquad\qquad \wedge \ w = xyz \\
\qquad\qquad \wedge \ y \neq \varepsilon \\
\qquad\qquad \wedge \ |xy| \leq n \\
\qquad\qquad \wedge \ \forall k \geq 0 \ : \ xy^k z \in L.
\end{array}
$$

Then we also have a theorem $\neg RHS \Rightarrow \neg Regular(L)$. This is how the Pumping Lemma works. Now, $\neg RHS$ for us is:

$$
\begin{array}{l}
\forall n \in N : \\
\exists w \in L_{suspect} \ : |w| \geq n \\
\qquad\qquad \wedge \\
\qquad\qquad \forall x, y, z \in \Sigma^* \ : \\
\qquad\qquad \vee \ w \neq xyz \\
\qquad\qquad \vee \ y = \varepsilon \\
\qquad\qquad \vee \ |xy| > n \\
\qquad\qquad \vee \ \exists k \geq 0 \ : \ xy^k z \notin L_{suspect}.
\end{array}
$$

Notice that we have to pick all x, y, z.

Revisit U versus Y (Asg8 Solution posted online)

The basicproblem was to show that

$$L_{uneq} = \{0^p 1^q \ \mid \ p \neq q\}$$

is not regular.

An **INCORRECT PROOF** would be as follows.

- Let there be an m-state DFA
- Pick a string $0^m 1^{m+1}$
- This is in L_{uneq}
- Pick y to be the string "0" i.e. of length 1
- Then when we pump up, we get $0^{m+1} 1^{m+1}$ which is not in L_{uneq}

- Problem with this proof: we DID NOT establish $\neg RHS$. For $\neg RHS$, we need to show for *all* y. In this selection, if $y =$ "00" were chosen, pump up would lead to $0^{m+2}1^{m+1}$ which still is in L_{uneq}
- So the "U versus Y" argument picked $0^m1^{m+m!}$ that is guaranteed to have a pump within 0 such that we will get to $0^{m+m!}1^{m+m!}$.

Revisit U versus Y (Asg8 Solution posted online)

A niftier proof for L_{uneq} being non-regular:

- The complement of L_{uneq} is $\{0^n1^n \mid n \geq 0\} \cup (0+1)^*1(0+1)^*0(0+1)^*$ (see Asg8 solutions).
- Let us refer to $(0+1)^*1(0+1)^*0(0+1)^*$ by $L_{.1.0.}$ and to $\{0^n1^n \mid n \geq 0\}$ by L_{0n1n}.
- *Note that there is no overlap between L_{0n1n} and $L_{.1.0.}$.*
- Thus, by using the Immiscible Union theorem, if we prove that L_{0n1n} is non-regular, then the complement of L_{uneq} would be shown non-regular.
- But we know that L_{0n1n} is non-regular (easier PL proof).
- Finally, if the complement of L_{uneq} is non-regular, then L_{uneq} is also non-regular.

When do formal machines accept what?

Formal machines in automata theory have different acceptance conditions:

- DFA and NFA:
  - Accept by final state (the control must be in a final state)
  - They must fully read the input
- (N)PDA:
  - Accept by final state **or** accept by empty stack (you have to say which convention is being used in your PDA)
  - They must fully read the input
- TM:
  - Accept by final state
  - *No need to fully read the input*

Why does this difference exist between TMs and other machines? Well, TMs capture the notion of a *computation* where the notion of halting is undecidable. It also helps model computations where the input has "don't cares" that need not be read before a decision can be made. In any case, that's what it is. We will now study details pertaining to PDA.

PDA Acceptance: by Final State or Empty Stack There are two different notions of acceptance of a string by a PDA. According to the first, a PDA accepts a string when, after reading the entire string, the PDA is in a final state. According to the second, a PDA accepts a string when, after reading the entire string, the PDA has emptied its stack.

Acceptance by final state and empty stack The PDAs we've seen so far accept by final state. There is another useful notion: acceptance by empty stack. To highlight the difference, we call these languages $L(P)$ and $N(P)$ respectively. For such PDAs, a string w is in its language exactly when the input is entirely consumed *and* an empty stack results in doing so.

Conversion of P_1 to P_2 ensuring $L(P_1) = N(P_2)$ Given a PDA P_1 that accepts by final state, we can obtain a PDA P_2 that accepts by empty stack such that $N(P_2) = L(P_1)$, simply by ensuring that P_2 has an empty stack exactly when P_1 reaches a final state (for the same input w seen by both these PDAs).

Conversion of P_1 to P_2 ensuring $N(P_1) = L(P_2)$ Given a PDA that is defined according to the "accept by empty stack" criterion, how do we convert it to a PDA that accepts by final state? A simple observation tells us that the stack can become empty at any control state. Therefore, the trick is to start the PDA with a new bottom of stack symbol z_{00}. Under normal operation of the PDA, we do not see z_{00} on top of the stack, as it will be occluded by the "real" top of stack z_0. However, in any state, if z_{00} shows up on top of the stack, we add

a transition to a newly introduced final state qF. qF is the only final state in the new PDA. Hence, whenever the former PDA drains its stack, the new PDA ends up in state qF.

Direct Conversion of PDAs to CFGs This algorithm will be the subject of a homework. It will use the notion of acceptance by empty stack.

Arguing Consistency and Completeness (Asg 9)
Suppose we have the grammar:

```
S -> ( W S | e
W -> ( W W | )
```

An outline of consistency and completeness may be found in the book chapters online.

Closure and Decidability

In this section, we catalog the main results you should remember, plus some justifications. Details are omitted for now.

1. Given a CFG, it *is* decidable whether its language is empty. Basically, if you find that `S` is not generating, the language of the grammar is empty! It is the bottom-up marking algorithm discussed above.
2. Given a CFG, it is *not decidable* whether its language is Σ^*.
3. The equivalence between two CFGs is not decidable. This follows from the previous result, because one of the CFGs could easily be encoding Σ^*.
4. Given a CFG, whether the CFG is ambiguous is not decidable. A standard reduction is *from* the Post Correspondence Problem.
5. Given a CFG, whether the CFG generates a *regular* language is not decidable.
6. CFLs are closed under union, concatenation, and starring because these constructs are readily available in the CFG notation.
7. CFLs are closed under reversal because we know how to "reverse a CFG."
8. CFLs are *not* closed under complementation, and hence also not closed under intersection.
9. CFLs are closed under intersection with a *regular language*. This is because we can perform the product state construction between a PDA and a DFA.
10. CFLs are closed under homomorphism.

Some Important Points Visited

We know that if L is a regular language, then L is a context-free language, but not vice versa. Therefore, the *space* of regular languages is *properly contained* in the space of context-free languages. We note some facts below:

- It **does not** follow from the above that the union of two CFLs is always a *non-regular* CFL; it is *not* so, in general. Think of $\{0^n 1^n \mid n \geq 0\}$ and the complement of this language, both of which are context-free, and yet, their union is Σ^* which is context-free, *but also regular*.
- The union of a context-sensitive language and a context-free language can be a regular language. Consider the languages L_{ww} and $\overline{L_{ww}}$ of Section **??**.

All this is made clear using a real-world analogy (see book chapters online).

Lost Venus Probe

Hoare [**?**] cites the story of a Venus probe that was lost in the 1960's due to a FORTRAN programming error. The error was quite simple - in hindsight. Paraphrased, instead of typing a "DO loop" as

```
DO 137 I=1,1000
...
137 CONTINUE,
```

the programmer typed

```
DO 137 I=1 1000
...
137 CONTINUE.
```

The missed comma caused FORTRAN to treat the first line as the assignment statement `DO137I=11000` — meaning, an assignment to a newly introduced variable `DO137I`, the value `11000`. The DO statement essentially did not loop 1000 times as was originally intended! FORTRAN's permissiveness was quickly dispensed with when the theory of context-free languages lead the development of "Algol-like" block-structured languages.

Sarcastically viewed, progress in context-free languages has helped us leapfrog into the era of deep semantic errors in programs, as opposed to unintended simple syntactic errors that caused programs to crash. The computation engineering methods discussed in later chapters in this book do help weed out semantic errors, which are even more notoriously difficult to pin down. We hope for the day when even these errors appear to be as shallow and simpleminded as the forgotten comma.

The best for the last

$L_{ww} = \{ww \mid w \in \{0,1\}^*\}$ is not context-free. We will show it (after midterm-2) using the CFL Pumping Lemma. However, its complement is! Let's derive its CFG! (It's in the book and also in notes19 as a practice problem).

Tips for Midterm-2

Please expect these concepts to be examined (a good checklist for you):

- [ ] Know how to tell between purely left-linear, purely right-linear, and the dangers of mixed linearity.
- [ ] Know how to reverse a CFG
- [ ] Get the Pumping Lemma facts straight
- [ ] Be able to design simple PDAs and CFGs
- [ ] Converting CFG to PDA (not the other way)
- [ ] Consistency and completeness of CFGs with respect to a language
- [ ] Ambiguity, inherent ambiguity. Also know where `if E then Matched else Unmatched` is needed (think of a simple sentence; it was a class question; the answer is not hard)
- [ ] Be able to design simple Turing machines
- [ ] Closure results and how to use them
- [ ] Be able to classify languages as being regular or context-free or non context-free
- [ ] Know about the Chomsky Hierarchy (notes19)
- [ ] Know where non-determinism makes a difference