CS 3100 – Models of Computation – Fall 2011 – Notes for L18

A CFG is a structure (N, Σ, P, S) where N is a set of symbols known as *non-terminals*, Σ is a set of symbols known as *terminals*, $S \in N$ is called the *start symbol*, and P is a finite set of *production rules* of the form $L \rightarrow R_1\ R_2\ \ldots R_n,$.

A *sentential form* is a string over $N \cup \Sigma$, and is obtained by starting from S and applying production rules to expand S. If we get rid of all of N, we call the resulting sentential form a *sentence*. Each such expansion step is called a *derivation step*. Sequences of derivation steps are called a *derivation sequence*. A *leftmost derivation sequence* is one where the leftmost non-terminal is expanded at every stage. The language of a CFG can be defined as follows:

$$\mathcal{L}(G) = \{w \mid S \Rightarrow^* w \ \wedge \ w \in \Sigma^*\}.$$

Let us denote a derivation step by $\Rightarrow$ and a derivation sequence by $\Rightarrow^*$. Consider grammar G_4 with these productions: $S \rightarrow SS \mid 1 \mid 0$. Then, a string 110 can now be derived in two ways:

- Through the leftmost derivation $S \Rightarrow SS \Rightarrow 1S \Rightarrow 1SS \Rightarrow 11S \Rightarrow 110$, or
- Through the rightmost derivation $S \Rightarrow SS \Rightarrow S0 \Rightarrow SS0 \Rightarrow S10 \Rightarrow 110$.

These yield different parse trees. This is an *ambiguous* grammar. An *ambiguous* grammar is one for which there is *at least one sentence* for which there exist two distinct parse trees. Since parse trees ascribe meanings to sentences, we don't want to admit even one sentence with two meanings (two parse trees)—hence the importance of this concept.

The interesting thing is that there exist *inherently ambiguous languages*—languages for which all CFGs are ambiguous. Here is an example:

$$L = \{a^i b^j c^k \mid i = j \ \vee \ j = k\}$$

Let us design one CFG for this language. The approach to designing a CFG is similar to how you program recursively: break the language description into simpler recursive cases that are glued together using union and concatenation. Each time you erect a pattern, you "grow" the language inside-out:

```
Language aEQbORbEQc:

S -> Mab Cs | As Mbc

Mab -> a Mab b | epsilon

Mbc -> b Mbc c | epsilon

Cs -> c Cs | epsilon

As -> a As | epsilon
```

One way to understand how these grammars "work" is by seeing how to build *push-down automata* which are machines that describe context-free languages.

Before we present a PDA for the above language `aEQbORbEQc`, here are some standard definitions, and some theorems:

- A language is said to be *context-free* if there is a context-free grammar describing it.

- Equivalently, a language is said to be *context-free* if there is a push-down automaton describing it.
- A CFG can be converted into a language-equivalent PDA
- A PDA can be converted into a language-equivalent CFG
- NPDA are different from DPDA; non-determinism makes a difference
- CFGs ("NCFG") are non-equivalent to DCFGs
- CFLs are closed under union, concatenation, Kleene-star, and reversal
- CFLs are *not* closed under intersection or complementation

0.1 First Attempt at a PDA for `aEQbORbEQc`

0.2 Second Attempt at a PDA for `aEQbORbEQc`

0.3 Standard CFG to PDA Conversion Illustrated on aEQbORbEQc CFG

1 Consistency, Completeness, Redundancy

Now consider grammar G_5 with production rules

Grammar G_5:
$S \rightarrow aSbS \mid bSaS \mid \varepsilon.$

The terminals are $\{a, b\}$. What CFL does this CFG describe? It is easy to see that in each replacement step, an S is replaced with either ε or a string containing an a and a b; and hence, all strings that can be generated from G_5 have the same number of a's and b's. Can *all* strings that contain equal a's and b's be generated using G_5? We visit this (much deeper) question in the next section. If you try to experimentally check this conjecture out, you will find that no matter what string of a's and b's you try, you can find a derivation for it using G_5 so long as the string has an equal number of a's and b's.

Note: We employ ε, `e`, and `epsilon` interchangeably, often for the ease of type-setting.

Consider the following CFG G_6 which has one extra production rule compared to G_5:

Grammar G_6:
$S \rightarrow aSbS \mid bSaS \mid SS \mid \varepsilon.$

As with grammar G_5, all strings generated by G_6 also have an equal number of `a`'s and `b`'s. If we identify this property as *consistency*, then we find that grammars G_5 and G_6 satisfy consistency. What about completeness? In other words, will *all* such strings be derived? Does it appear that the production $S \rightarrow SS$ is essential to achieve completeness? It turns out that it is not - we can prove that G_5 is complete, thus showing that the production $S \rightarrow SS$ of G_6 is *redundant.*

How do we, in general, prove grammars to be complete? The general problem is undecidable,[1] However, for *particular* grammars and *particular* completeness criteria, we can establish completeness, as we demonstrate below.

[1]The undecidability theorem that we shall later show is that for an *arbitrary* grammar G, it is not possible to establish whether $L(G)$ is equal to Σ^*.

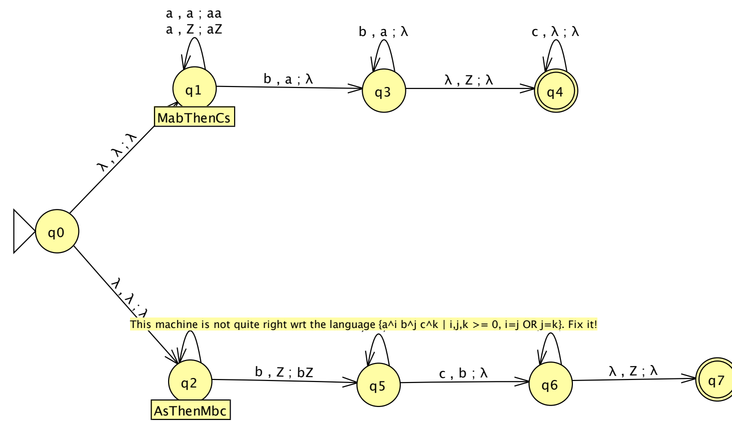

Figure 1: An incorrect PDA for aEQbORbEQc

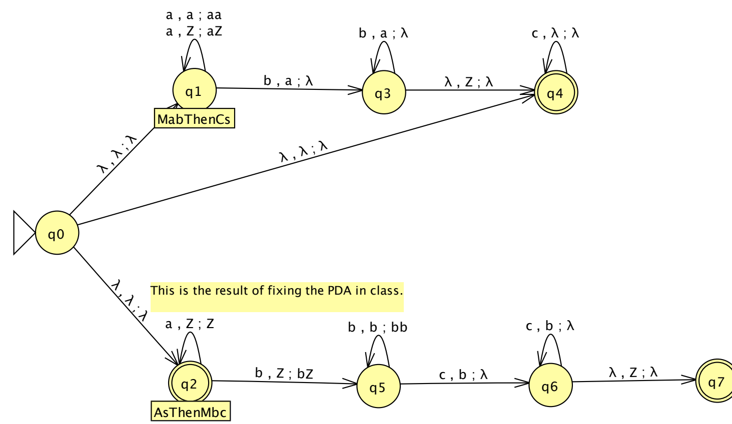

Figure 2: Corrected PDA for aEQbORbEQc

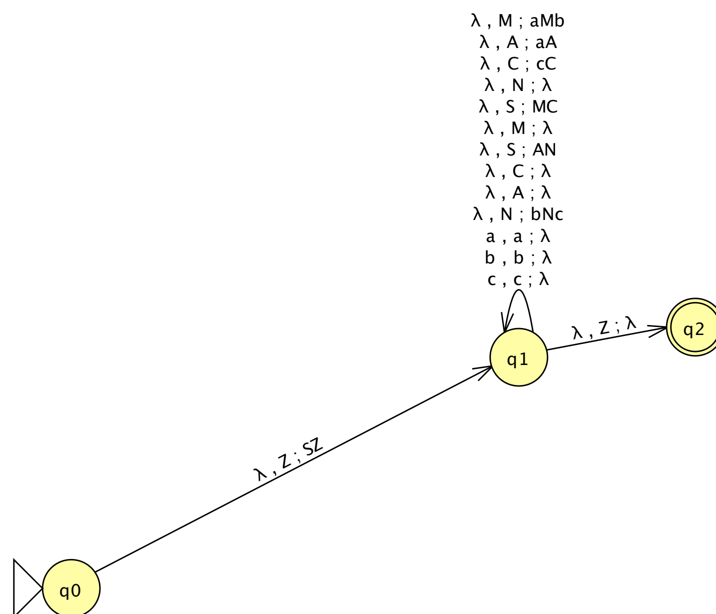

Figure 3: PDA obtained by direct conversion from CFG

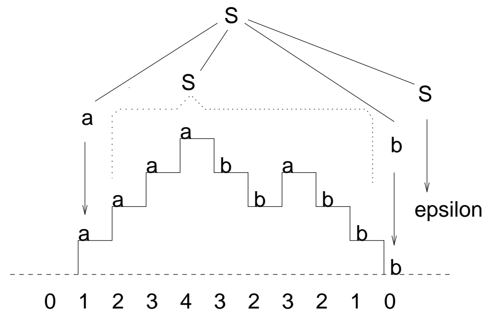

Figure 4: A string that does not cause zero-crossings. The numbers below the string indicate the running difference between the number of a's and the number of b's at any point along the string

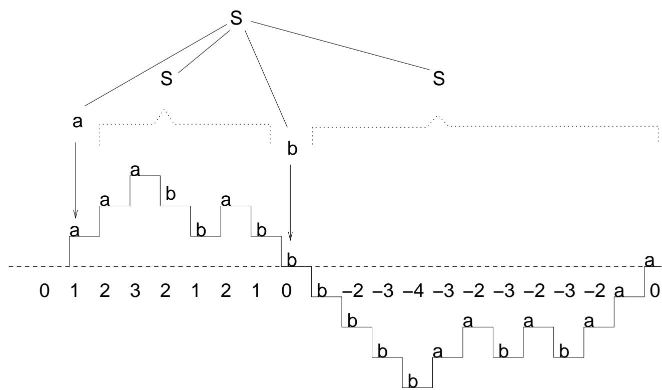

Figure 5: A string that causes zero-crossings

Proof of completeness: The proof of completeness typically proceeds by induction. We have to decide between arithmetic or complete induction; in this case, it turns out that complete induction works better. Using complete induction, we write the *inductive hypothesis* as follows:

> Suppose G_5 generates all strings less than n in length having an equal number of `a`'s and `b`'s.

Consider now a string of length $n + 2$ – the next longer string that has an equal number of `a`'s and `b`'s. We can now draw a graph showing the running difference between the number of `a`'s and the number of `b`'s, as in Figure 3 and Figure 4. This plot of the running difference between `#a` and `#b` is either fully above the x-axis, fully below, or has zero-crossings. In other words, it can have many "hills" and "valleys." Let us perform a case analysis:

1. The graph has no zero-crossings. There are further cases:
   (a) it begins with an `a` and ends with a `b`, as in Figure 3.
   (b) it begins with a `b` and ends with an `a` (this case is symmetric and hence will not be explicitly argued).
2. It has zero-crossings, as in Figure 4. Again, we consider only one case, namely the one where the first zero-crossing from the left occurs after the curve has grown in the positive direction (i.e., after more `a`'s occur initially than `b`'s).

Let us consider case 1a. By induction hypothesis, the shorter string in the middle can be generated via `S`. Now, the entire string can be generated as shown in Figure 3 using production `S -> aSbS`, with `a` matching the first `a`, the first `S` matching 'the shorter string in the middle,' the `b` matching the last `b` in the string, and the second `S` going to ε. Case 1b may be similarly argued. If there is a zero-crossing, then we attack the induction as illustrated in Figure 4, where we split the string into the portion before its last zero-crossing and the portion after its last zero-crossing. These two portions can, by induction, be generated from G_5, with the first portion generated as `aSb` and the second portion generated as an `S`, as in Figure 4.

Illustration 1.1 *Consider*

$$L_{a^m b^n c^k} = \{a^m b^n c^k \mid m, n, k \geq 0 \textit{ and } ((m = n) \textit{ or } (n = k))\}$$

Develop a context-free grammar for this language. Prove the grammar for consistency and completeness.
Solution: The grammar is given below. We achieve "equal number of `a`'s and `b`'s" by growing "inside out," as captured by the rule `M -> a M b`. We achieve zero or more `c`'s by the rule `C -> c C` or `e`. Most CFGs get designed through the use of such "idioms."

```
S -> M C | A N

M -> a M b | e

N -> b N c | e

C -> c C | e

A -> a A | e
```

Consistency: No string generated by `S` must violate the rules of being in language $L_{a^m b^n c^k}$. Therefore, if `M` generates matched a's and b's, and `C` generates only c's, consistency is guaranteed. The other case of `A` and `N` is very similar.

Notice that from the production of `M`, we can see that it generates matched a's and b's in the `e` case. Assume by induction hypothesis that in the `M` occurring on the right-hand side of the rule, `M -> a M b`, respects consistency. Then the `M` of the left-hand side of this rule has an extra `a` in front and an extra `b` in the back. Hence, it too respects consistency.

Completeness: We need to show that any string of the form $a^n b^n c^k$ or $a^k b^n c^n$ can be generated by this grammar. We will consider all strings of the kind $a^n b^n c^k$ and develop a proof for them. The proof for the case of $a^k b^n c^n$ is quite similar and hence is not presented.

We resort to arithmetic induction for this problem. Assume, by induction hypothesis that the particular $2n+k$-long string $a^n b^n c^k$ was derived as follows:

- `S` $\Rightarrow$ `M C`.
- `M` $\Rightarrow^*$ $a^n b^n$ through a derivation sequence that we call S1, and
- `C` $\Rightarrow^*$ c^k through derivation sequence S2.
- `S` $\Rightarrow$ `M C` $\Rightarrow^*$ $a^n b^n$ `C` $\Rightarrow^*$ $a^n b^n$ c^k. Notice that in this derivation sequence, the first $\Rightarrow^*$ derivation sequence is what we call S1 and the second $\Rightarrow^*$ derivation sequence is what we call S2.

Now, consider the next legal longer string. It can be either $a^{n+1} b^{n+1} c^k$ or $a^n b^n c^{k+1}$. Consider the goal of deriving $a^{n+1} b^{n+1} c^k$. This can be achieved as follows:

- `S` $\Rightarrow$ `M C` $\Rightarrow$ `a M b C`.
- Now, invoking the S1 sequence, we get $\Rightarrow^*$ a $a^n b^n$ b `C`.
- Now, invoking the S2 sequence, we get $\Rightarrow^*$ a $a^n b^n$ b c^k; and hence, we can derive $a^{n+1} b^{n+1} c^k$.

Now, $a^n b^n c^{k+1}$ can be derived as follows:

- `S` $\Rightarrow$ `M C` $\Rightarrow$ `M c C`.
- Now, invoking the S1 derivation sequence, we get $\Rightarrow^*$ $a^n b^n$ `c C`.
- Finally, invoking the S2 derivation sequence, we get $\Rightarrow^*$ $a^n b^n$ c^{k+1}.

Hence, we can derive any string that is longer than $a^n b^n c^k$, and so by induction we can derive all legal strings.