§1 05-07

Previously we were working with finite memory machines. Limitations on what we could do even when we came to omega languages.

Two ways to recognize regular languages. Regular languages in terms of deterministic finite automata, and regular expressions.

Now we are getting infinite memory. A finite state machine plus an auxiliary stack. Then in the next module we will get two stacks.

§1.1 Context Free Languages

We will define CFL's in terms of grammars.

We will define a grammar as a way to generate strings in the language. There is a lot more structure than regex had. Rules for producing strings.

Definition 1.1 (Context-free Grammar)**.** A context-free grammar consists of

1. a set of symbols called terminals (Σ alphabet)
2. Another set of symbols called non-terminals or variables V.
3. Both V, Σ are finite, and $V \cap \Sigma = \varnothing$.
4. A special variable called the start symbol, usually but not always S.
5. A set of rules called PRODUCTIONS.

$$A \to \alpha,\ A \in V,\ \alpha \in (V \cup \Sigma)^*$$

Example 1.2

Let $\Sigma = \{a, b\}$. These are our terminal symbols. Terminal suggests that something has to be terminated. Variable on the other hand suggests something that can change. $V = \{S\}$. Just the start symbol. Rules:

$$S \to \epsilon,\ S \to aSb$$

Apply any rule you like to replace a non-terminal with the RHS of a rule. It doesn't matter what the surrounding context is. This is why it's called context free.

$$S \to aSb \to aaSbb \to aaaSbbb \to aaabbb$$

This allows you to create $\{a^n b^n \mid n \geq 0\}$, which is not a regular language. CFL's are clearly more powerful than regular languages.

When you obtain a string with no more non-terminals, you have generated a string in the context free language.

$$L(G) = \{w \in \Sigma^* \mid w \text{ can be generated from the start symbol } S\}$$

Unlike with DFA the language is defined and then you have to recognize it, here you are generating the language.

Example 1.3 (Grammar for arithmetic expressions)

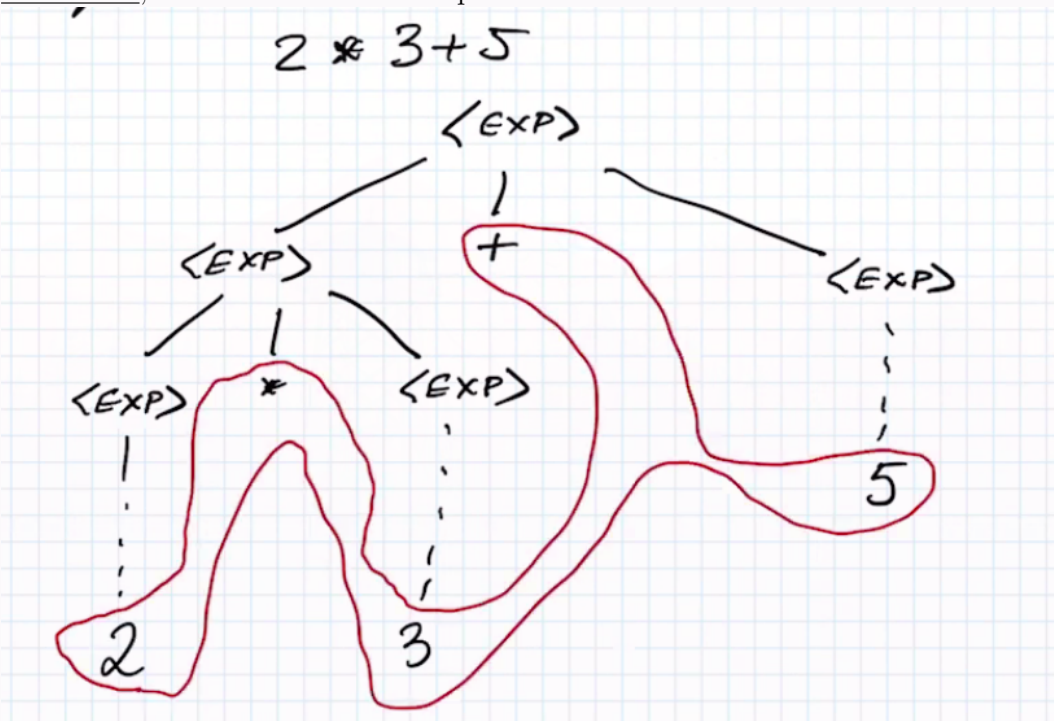
All modern programming languages, the syntax is a context free language, oftentimes it is even more restricted than that. Cobalt does not have a context free grammar, but now we've realized that this is a stupid idea. Every modern programming language is context free.

$$\Sigma = \{0, 1, \cdots, 9, \times, +, (,)\}$$
$$V = \{\langle EXP\rangle, \langle NUM\rangle, \langle NZ\rangle, \langle N\rangle\}$$

Start symbol $< EXP >$. Notation to save space.

$$\begin{aligned}
\langle EXP\rangle &\to \langle EXP\rangle + \langle EXP\rangle \mid \langle EXP\rangle \times \langle EXP\rangle \mid \langle EXP\rangle \mid \langle NUM\rangle \\
\langle NUM\rangle &\to 0 \mid \langle NZ\rangle \\
\langle NZ\rangle &\to 1\langle N\rangle \mid 2\langle N\rangle \mid \cdots \mid 9\langle NZ\rangle \\
\langle N\rangle &\to 0\langle N\rangle \mid 1\langle N\rangle \mid 2\langle N\rangle \mid \cdots \mid 9\langle NZ\rangle \mid \epsilon
\end{aligned}$$

NZ is to allow you to write zero but prevent you from writing 00. To properly display derivations, we will use trees called parse trees.



Definition 1.4 (Ambiguity in Context Free Languages)**.** The grammar allows two (or more) distinct ways of parsing an expression. Such a grammar is called ambiguous. An example of bad grammar engineering. A given language does not have a unique grammar. With regular languages there was a unique minimal automaton; there is no such concept in context free languages.

It is possible to redesign the grammar so that it is not ambiguous, at least in this case. There exist languages for which no unambiguous language is possible. Then it is inherently ambiguous.

§1.2 Designing CFG

Example 1.5 (Counting CFG)
How to produce the language $L = \{a^n b^{2n} \mid n \geq 0\}$. Rules:

$$S \to aSbb \mid \epsilon$$

Keep going until you are happy and then wipe out the S. Notice that context free languages can count

Example 1.6 (Palindromes)
$L = \{x \in \Sigma^* \mid x = x^{REV}\}$. Rules:

$$S \to aSa \mid bSb \mid a \mid b \mid \epsilon$$

Example 1.7
$L = \{x \in \Sigma^* \mid \#_a(x) = \#_b(x)\}$. A bit harder than example number 1. To make it easier let $d(x) = \#_b(x) - \#_a(x)$ and $L = \{x \mid d(x) = 0\}$.

Let $x \in L$, and u be the shortest prefix of x such that $d(u) = 0$ and suppose u starts with b. u must end with a because it starts with b. So $u = bva$ and $d(v) = 0$.

$x = uz \Rightarrow d(z) = 0$

$$S \to \epsilon \mid bSaS \mid aSbS$$
alternatively
$$S \to aSb \mid bSa \mid SS \mid \epsilon$$

Fact 1.8. Every regular language is a CFG.

Definition 1.9. A CFG is said to be in Chomsky Normal Form if every rule has the form

$$A \to BC \text{ or } A \to a$$

Here A, B, C are non-terminals and a is a terminal. We do not allow rules like $A \to \epsilon$ except for the start symbol. Some resources say that even the start symbol can't go to epsilon.

Theorem 1.10

Every context free language is generated by a context free grammar in chomsky normal form.

Proof. The idea is to systematically get rid of bad rules. e.g. $A \to \epsilon$ is a forbidden rule so throw it out and look at any rule with A on the RHS e.g. $B \to \alpha_1 A \alpha_2 A$. Add all possible rules with some of these A's removed. So in this case add the following rules.

$$B \to \alpha_1 \alpha_2 A$$
$$B \to \alpha_1 A \alpha_2$$
$$B \to \alpha_1 \alpha_2$$

Many other modifications of this type. Consider

$$A \to x_1 x_2 x_3 \cdots x_n, \ x_i \in \Sigma \cup V$$
becomes
$$A \to x_1 A_1, x_1 \to x_1 \text{ if } x_1 \in \Sigma$$
$$A_1 \to x_2 A_2, \ \ldots$$

All these A_i are new non terminals. □

You can see that there are lots of ways to modify the grammar while preserving the languages.

§1.3 Closure Properties of CFL's

1. L_1, L_2 are CFL's then $L_1 \cup L_2$ is a CFL.

   Let S_1 be the start for G_1 and $L(G_1) = L_1$, and S_2 be the start for G_2 and $L(G_2) = L_2$. Take all the rules together and add $S \to S_1 \mid S_2$.

2. $L_1 \cdots L_2$ is also a context free language.

   $$\to S_1 S_2$$

3. L^* is a context free language if L is. Add a new start symbol S' and add the rule

   $$S' \to SS' \mid \epsilon$$

4. CFL's are <u>not</u> closed under

   a) intersection

   b) complement

Easy proofs becaues of the power of grammars rather than fiddling with automata.

Example 1.11

$L_1 = \{a^n b^n c^m \mid n, m \geq 0\}$ is a CFL, and so is $L_2 = \{a^m b^n c^n \mid n, m \geq 0\}$. Then $L_1 \cap L_2 = \{a^n b^n c^n \mid n \geq 0\}$. The idea is that when there is a buffer, you can't count. This is our favorite example of something that is not context free.

Tomorrow we will see a new pumping lemma for context free languages.

Note also that $L = \{a^n b^n c^n \mid n \geq 0\}$ is not a CFL but $\overline{L}$ is a CFL! Also see that if it were closed under complementation and under union it would have been closed under intersection. So it cannot possibly be closed under complementation.

Proposition 1.12

If L is a CFL and R is a regular language then $L \cup R$ is a CFL.

Fact 1.13. There is an algorithm to look at a grammar and decide if $L(G) = \varnothing$. It's easy to write grammars that don't produce anything. For example if you don't give a rule for the start symbol. No matter how many other rules you have it won't produce anything. It might be that you never get rid of the non-terminals, which by definition are not in the language. The point is that there is an algorithm that can look and decide.

Proof. DIY □

Algorithm to decide if $w \in L(G)$. Works on general context free grammar (even ambiguous ones). To reduce the amount of book keeping assume G is in chomsky normal form.

Typical dynamic programming algorithm. Given $G = (V, \Sigma, S, P)$. P for the set of rules because they are referred to as productions. Input $w = a_1 \cdots a_n \in \Sigma^*, a_i \in \Sigma$.

The idea is to work bottom up to construct a possible derivation for w. To get a specific letter a we must have used a rule of the form $A \to a$.

We define inductively a 2-indexed family of subsets of V.

$$i \leq j,\ X_{ij} \coloneqq \{A \in V \mid A \to a_i \cdots a_j\}$$
$$X_{ii} = \{A \in V \mid A \to a_i\}$$

When we come to compute X_{ij}, we assume that we have already computed X_{ij}, X_{kj} for all k between i and j When we come to compute X_{ij}, we assume that we have already computed X_{ik}, X_{kj} for all k between i and j. So if $B \in X_{ik}$, $C \in X_{(k+1)j}$ and $A \to BC$ is a rule in G then we put $A \in X_{ij}$.

§1.4 Algorithms For CFLs

1. $L(G) = \varnothing$ is decidable.
2. $L(G)$ infinite is decidable.

3. $L(G) = \Sigma^*$ is undecidable. This is surprising at first. Provably impossible.

Start of a topic called parsing. Heavily studied in a compiler class. Reached such a high state of understanding that nobody really needs to write parsers anymore because people have written parser generators.

§1.5 Pushdown Automata

This is an NFA + Stack (1 stack only, no more). Emphasis on NFA. Nondetermism is built in to the structure. DFA pushdown automata are not equivalent to NFA pushdown automata. There is an equivalent deterministic context free language.

> **Theorem 1.14**
> Every CFL can be recognized by a PDA (pushdown automata) and every language recognized by a PDA is a CFL. A language is context free if and only if it is recognized by a PDA.
>
> Use judgement. Sometimes it is much easier to do a proof via PDA, and sometimes easier via grammar.

Definition 1.15 (Pushdown Automata)**.** Q represents the states (finite).

Σ represents the input alphabet (finite). $\Sigma_\epsilon = \Sigma \cup \{\epsilon\}$.

γ represents the stack alphabet $\gamma \supset \Sigma$, $\gamma_\epsilon = \gamma \cup \{\epsilon\}$

$\delta : Q \times \Sigma_\epsilon \times \gamma_\epsilon \to P_f(Q \times \gamma_\epsilon)$

P_f is the finite power set.

Reading a letter involves (i) looking at the input and the top of the stack, (ii) changing the state, (iii) poping the stack or pushing something on the stack or both and then move to the next input symbol.

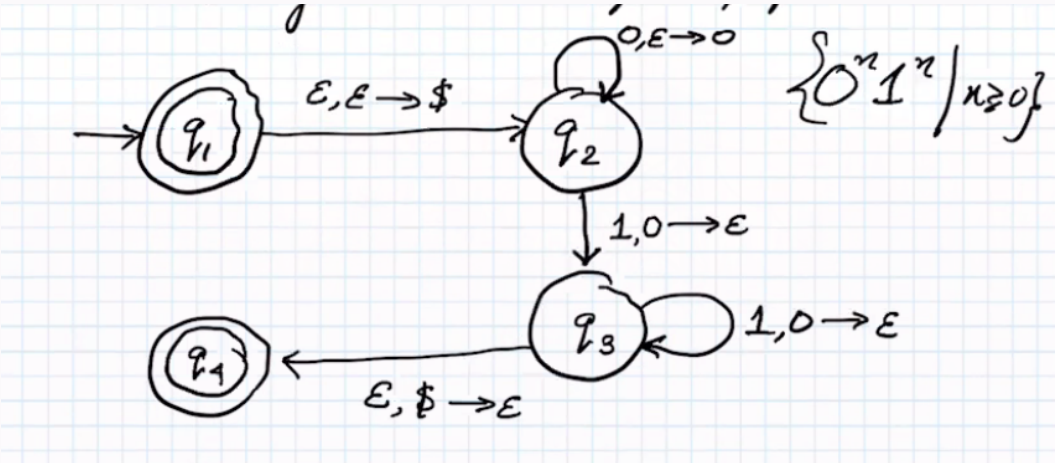
Notation, $a, b \to c$. b is seen on top of the stack, a is the next input symbol, pop the stack and replace b with c. a may be ϵ which would mean don't read input

Reading a letter involves (i) looking at the input and the top of the stack, (ii) changing the state, (iii) poping the stack or pushing something on the stack or both and then move to the next input symbol.

Notation, $a, b \to c$. b is seen on top of the stack, a is the next input symbol, pop the stack and replace b with c.

a may be ϵ which would mean don't read input. b may be ϵ which would mean just push c. c may be ϵ which would mean just pop the stack.

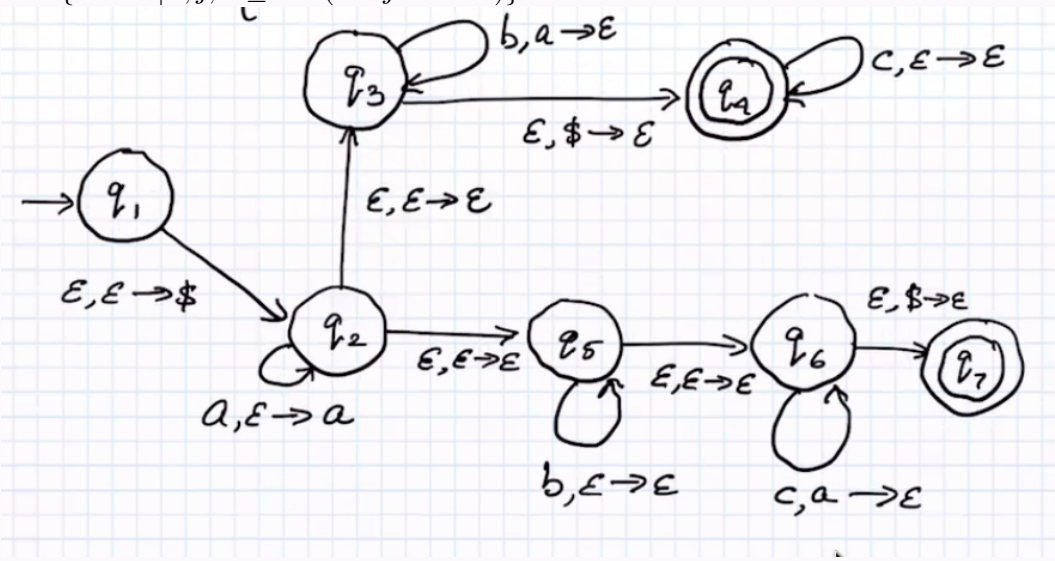
Example 1.16



You are required to go all the way to the end of the input or you are jammed and rejected. This is an example of a deterministic machine but in general you can have non determinism.

Example 1.17

$L = \{a^i b^j c^k \mid i, j, k \geq 0 \wedge (i = j \vee i = k)\}$



Note 1.18.

1. Acceptance only happens at the end
2. A PDA cannot "decide to jam" when there are valid moves.

Note 1.19. There is an alternative notion of acceptance: no accept states, we accept if the stack is empty at the end of the input. These are equivalent.

$a^n b^n c^n$ cannot be accepted by a PDA because we can the a's and count off the b's

but then we don't know what to do with the c's. If we had 2 stacks, we could accept $\{a^n b^n c^n \mid n \geq 0\}$ by pushing to the second stack as we pop from the first stack.

$\{a^n b^n c^n d^n \mid n \geq 0\}$ still only needs 2 stacks, because you can go back and forth between the two stacks. 2 stacks is enough for any number.

Fact 1.20. A PDA with two stacks is universal. i.e. it has the same power as a turing machine and can compute any computable function.

People realized that complexity is more significiant than power, because the power of everything is the same once you get to two stacks, two queues, etc.