DISCLAIMER: These notes are not necessarily an accurate representation of what I said during the class. They are mostly what I intend to say, and have not been carefully edited.

- Comments from last time: 2 STS, 2 JR, 1 STF.
  1. Erasing notes too quickly.
  2. CFGs and parse trees.
- Today’s topics:
  1. Context free grammars.
  2. Chomsky Normal Forms.
  3. Quiz 5.
  4. Pushdown automatas.

Last time we introduced the notion of context free grammars. Plan for today is to go through a few more examples, as well as two alternate ways of representing them.

Recall that a context free grammar (CFG) is a 4-tuple $(V, \Sigma, R, S)$ where:

1. $V$ is the variables.
2. $\Sigma$ is a finite set, disjoint from $V$, known as the terminals.
3. $R$ is a set of rules, each rule being the form $V \rightarrow \{V \cup \Sigma\}^*$
4. $S \in V$ is the starting variable.

For strings of variables and terminals $uvw$ such that $A \rightarrow w$ is a rule, we say that $uAv$ yields $uvw$, written as $uAv \Rightarrow ufwv$

We can compose such steps to form derivations. Specifically we say $u \Rightarrow^* v$
if there is a sequence of strings (of variables and terminals) \( v_1 \ldots v_k \) such that
\[
u \Rightarrow v_1 \Rightarrow v_2 \Rightarrow \ldots \Rightarrow v_k \Rightarrow v_k.
\]

Such sequence of derivations produce parse trees: each variable ‘blows up’ to a word consisting of terminal symbols by invoking the rules (possibly many times).

Such parse trees are particularly useful in programming languages since they also denote an order of applying the operations: the topmost variable is the one applied last.

For example, if we want to describe expressions involving + and \( \times \), as well as the variables \( a, b, \) and \( c \) (note that these are really terminals in the CFG grammar sense), we can create the grammar:

\[
\langle \text{Expr} \rangle \rightarrow \langle \text{Expr} \rangle + \langle \text{Term} \rangle | \langle \text{Term} \rangle \\
\langle \text{Term} \rangle \rightarrow \langle \text{Term} \rangle \times \langle \text{Factor} \rangle | \langle \text{Factor} \rangle \\
\langle \text{Factor} \rangle \rightarrow (\langle \text{Expr} \rangle) | a | b | c
\]

Here we can then check that \( a + b \times c \) and \((a + b) \times c\) parses the way that we normal think about arithmetic operations.

1 Chomsky Normal Form

Chomsky norm form is a variant where the rules are more limited. That is, the only three types of rules allowed are:

1. \( A \to BC \) where \( B \) or \( C \) are variables (not terminals), and cannot be the start variable.
2. \( A \to a \) where \( a \) is a terminal (that’s not \( \epsilon \)).
3. \( S \to \epsilon \).

For example, recall the CFG for strings with the same numbers of 0s as 1s from last time:

\[
S \to 0S1 | \epsilon
\]

To convert this into Chomsky normal form, note that \( S \to \epsilon \) is still allowed. So we create a new variable for the first case \((0S1)\)

\[
S \to A \\
S \to \epsilon.
\]

Then we need to convert \( A \). For this we use a variable \( B \) to represent the first 0, \( C \) to represent the second half, and \( D \) to represent the last 1. They give rise to the rules:

\[
A \to BC \\
B \to 0 \\
C \to S \\
D \to 1
\]
Theorem 1.1. Every CFG can be converted to an equivalent CFG that’s in Chomsky normal form.

2 Pushdown Automatas

The punchline for the next three lectures is that CFGs are equivalent to DFAs equipped with an additional stack.

Stacks have infinite storage, so in a sense allows us to ‘count’ the number of things we’ve encountered. They actually allow us to do more than that: we can track, in reverse order, the important symbols that we encountered.

Recall the non-regular language that got us started

\[ \{0^n1^n | n \geq 0\} \]

If we have a DFA with a stack, we will put every 0 that we see onto the stack. After we encounter a 1, we only accept 1s, until the stack runs out of them.

Formally, the pushdown automata (PDA) has on extra type of symbols: the stack symbols \( \Gamma \). Its states \( Q \), alphabet \( \Sigma \), starting state \( q_0 \), and accepting states \( F \subseteq Q \) are still the same as before. However its transitions are now

\[ Q \times \Sigma_e \times \Gamma_e \rightarrow \mathcal{P} (Q \times \Gamma_e) \]

That is, the transitions are non-deterministic, and each transition consumes a stack symbol (which is possibly \( \epsilon \)), and adds a symbol to the stack (that’s also possibly \( \epsilon \)).

Continuing with the example with the language

\[ \{0^n1^n | n \geq 0\} \]

a main difficulty is that we need to ensure that we need to finish with the stack empty.

To do so, we create a new symbol \( \$ \), that we push onto the stack at the beginning, and we push it off to finish. This leads to four states \( q_0, q_1, q_2, q_3 \), with \( q_1 \) and \( q_2 \) being the states that we consume 0s and 1s at respectively.

Then:

1. the transition from \( q_0 \) to \( q_1 \) consumes no input, and adds \( \$ \) to the stack,
2. \( q_1 \) loops onto itself by consuming 0s from the input, and pushing \( x \) onto the stack.
3. \( q_1 \) goes to \( q_2 \) by consuming a 1 from the input, as well as an \( x \) from the stack.
4. \( q_2 \) loops onto itself by consuming 1s from the input along with \( xs \) from the stack.
5. \( q_2 \) then goes to \( q_3 \) by consuming the final \( \$ \) from the stack, and nothing from the input.
Here $q_0$ and $q_3$ are accepting: $q_0$ corresponds to the empty string, and $q_3$ corresponds to what happens after we’ve pushed off everything.

To indicate these on the diagram, we use

$$a, \gamma_1 \rightarrow \gamma_2$$

To indicate we consume input $a$, remove $\gamma_1$ from the stack, and push $\gamma_2$ onto the stack.

If time permits, we will discuss one additional example of a PDA, for the language

$$\{a^i b^j c^k | i, j, k \geq 0 \land (i = j \lor i = k)\}.$$