Grammars

Thus far we have been doing a lot of computation. We have been doing arithmetic, finding squares, and even comparing numbers of symbols in strings. For example, in our study of automata, we found that pushdown machines could recognize strings of the form $a^n b^n$. This was done by counting the a's and then checking them off against the b's. Let's ask about something that must happen before we can recognize a string. How was that string generated in the first place? What kind of algorithm might be used to produce strings of the form $a^n b^n$? Consider the following computing procedure.

write down the symbol #
while the string is not long enough
keep replacing the # with the string a#b
replace the # with the string ab

This is a very simple procedure. We should be convinced that it does indeed produce strings of the form $a^n b^n$. Watch what happens when we execute the algorithm. We start with a #, then replace it with a#b, then it grows to aa#bb, and so forth until we've had enough. Then we just replace the # with a final ab and we have a string of a's followed by exactly the same number of b's.

If we analyze what took place, we discover that we were applying three rules in our algorithm. Those rules were:

a) start with a #,
b) replace the # with a#b, and
c) replace the # with ab.

We should note two things here. We begin with a # and end up by replacing it with the string ab. Then, when the # is gone no more rules apply. We could use a little shorthand notation (we'll use an arrow instead of the phrase gets replaced with) and write our rules as follows.

a) Start $\rightarrow$ #
b) # $\rightarrow$ a#b
c) # $\rightarrow$ ab
So far this looks pretty good. Now we get rid of the # and replace it by the capital letter A. Also, we represent the word Start by its first letter, the symbol S. Now we have this set of rules:

\[
\begin{align*}
S & \rightarrow A \\
A & \rightarrow aAb \\
A & \rightarrow ab
\end{align*}
\]

and these rules tell how to replace upper case letters with strings. If we are to use these rules to generate strings, the main algorithm becomes:

```plaintext
string = S
while capital letters exist in the string
apply an appropriate replacement rule
```

Note that this algorithm works for any group of rules like those provided above. We just replace a capital letter with the right hand side (the symbols following the arrow) of any of the rules in which it appears as the left hand side. This is fairly simple! Generating strings seems to be very easy with the proper set of rules.

Let's try generating a string. For example, aaabbb. Here are the steps:

\[
\begin{align*}
S & \\
A & \\
aAb & \\
aaAb & \\
aaaAb &
\end{align*}
\]

That's not too bad. We just apply the right rules in the proper order and everything works out just fine.

Let's try something not quite so easy. Let's take a close look at something important to the computer scientist - an arithmetic assignment statement. It is a string such as:

\[ x = y + (z \times x) \]

Let's consider how it was generated. It contains variables and some special symbols we recognize as arithmetic operators. And, in fact, it has the general form:

\[ <\text{variable}> = <\text{expression}> \]
So, the starting rule for these statements could be:

\[ S \rightarrow V=E \]

if we substitute \( V \) and \( E \) for \(<\text{variable}>\) and \(<\text{expression}>\). This is fine. But we are not done yet. There are still things we do not have rules on how to generate. For example, what's an expression? It could be something with an operator (the symbols + or \( \ast \)) in the middle. Or it could be just a variable. Or even something surrounded by parentheses. Putting all of this into rules yields:

\[
\begin{align*}
S & \rightarrow V=E \\
E & \rightarrow E+E \\
E & \rightarrow E*E \\
E & \rightarrow (E) \\
E & \rightarrow V
\end{align*}
\]

So far, so good. We're almost there. We need to define variables. They are just letters like \( x, y, \) or \( z \). We write this as

\[ V \rightarrow x \mid y \mid z \]

where the vertical line means or (so that \( V \) can be \( x \), \( y \), or \( z \)). This is a little more shorthand notation.

Note that there are several ways to generate our original arithmetic statement from the set of rules we have derived. Two sequences of steps that generate the string are:

\[
\begin{array}{ll}
S & S \\
V=E & V=E \\
x=E & V=E+E \\
x=E+E & V=E+(E) \\
x=V+E & V=E+(E*E) \\
x=y+E & V=E+(E*V) \\
x=y+(E) & V=E+(E*x) \\
x=y+(E*E) & V=E+(V*x) \\
x=y+(V*E) & V=E+(z*x) \\
x=y+(z*E) & V=E+(z*x) \\
x=y+(z*V) & V=y+(z*x) \\
x=y+(z*x) & x=y+(z*x)
\end{array}
\]

These sequences are called derivations (of the final string from the starting symbol \( S \)). Both of the above are special derivations. That on the left is a leftmost derivation because the capital letter on the left is changed according to some rule in each step. And, of course the derivation on the right is called a
rightmost derivation. Another way to present a derivation is with a diagram called a derivation tree. An example is:

```
S  =  E
  /\  /
 /  \ /  \  /
E   E  E
 +  *  +  *
  /  /  /  /
 /  /  /  /
E   E  E  E
  /  /  /  /
 /  /  /  /
V   V  V  V
```

We need to note here that derivations are done in a top-down manner. By this we mean that we begin with an S and apply rules until only small letters (symbols of our target alphabet) remain. There is also a bottom-up process named parsing associated with string generation that we shall investigate later. It is used when we wish to determine exactly how a string was generated.

As we have seen, sets of strings can be generated with the aid of sets of rules. (And, we can even reverse the process in order to determine just how the string was generated.) Grammars are based upon these sets of rules. Here is how we shall define them.

**Definition.** A grammar is a quadruple $G = (N, T, P, S)$ where:

- $N$ is a finite set (of nonterminal symbols),
- $T$ is a finite alphabet (of terminal symbols),
- $P$ is a finite set (of productions) of the form $\alpha \rightarrow \beta$
  - where $\alpha$ and $\beta$ belong to $(N \cup T)^*$, and
  - $S \in N$ (is the starting symbol).

At this point we should be rather comfortable with the definition of what a grammar is. Our last example involving assignment statements had $\{S, E, V\}$ as the nonterminal set and $\{x, y, z, +, *, (, ), =\}$ as its alphabet of terminals. The
terminals and nonterminals must of course be disjoint if we wish things to not get awfully confusing.

Generating strings with the use of grammars should also be a comfortable topic. (Recall that we begin with the starting symbol and keep replacing nonterminals until none remain.) Here is a precise definition of one step in this generating process.

**Definition.** Let $G = (N, T, P, S)$ be a grammar. Let $\alpha$, $\beta$, $\gamma$, and $\xi$ be strings over $N \cup T$. Then $\gamma\alpha\xi$ yields $\gamma\beta\xi$ (written $\gamma\alpha\xi \Rightarrow \gamma\beta\xi$) under the grammar $G$ if and only if $P$ contains the production $\alpha \rightarrow \beta$.

A derivation is just a sequence of strings where each string yields the next in the sequence. A sample derivation for our first example (strings of the form $a^n b^n$) is:

$$S \Rightarrow A \Rightarrow aAb \Rightarrow aaAbb \Rightarrow aaabbb.$$  

Our next definition concerns the sets generated by grammars. These are called the languages generated by the grammars.

**Definition.** The language generated by $G = (N, T, P, S)$ (denoted $L(G)$) is the set of all strings in $T^*$ which can be derived from $S$.

That is all there is to grammars and formal languages. Well, that is not quite so. We shall spend the remainder of the chapter examining their properties and relating them to the machines we have examined thus far. To begin this process, we will now set forth a hierarchy of languages developed by Noam Chomsky and thus called the Chomsky hierarchy.

First, the unrestricted grammars that are named type 0 grammars (with their companion languages named, the type 0 languages). Anything goes here. Productions are of the form mentioned in the definition. Right and left hand sides of productions can be any kind of strange combination of terminals and nonterminals. These are also known as phrase structure grammars.

Next, the type 1 grammars. These have much better behaved productions. Their right hand sides must not be shorter than their left hand sides. In other words, productions are of the form $\alpha \rightarrow \beta$ where the length of $\beta$ is at least as long as the length of $\alpha$. These are known as length preserving productions. The type 1 grammars and languages are called context sensitive because productions such as:

$$aAB \rightarrow abA$$
which depend upon context (note that AB cannot change to a bA unless an a is next to it) are allowed and encouraged.

Type 2 grammars are simpler still. The productions must be length preserving and have single nonterminal symbols on the left hand side. That is, rules of the form $A \to \alpha$. The examples at the beginning of the section are of this type. These are also known as the context free grammars and languages.

Last (and least) are the type 3 grammars. Not much is allowed to happen in their productions. Just $A \to dB$ or $A \to d$ where $d$ is a terminal symbol and both $A$ and $B$ are nonterminals. These are very straightforward indeed. They are called regular or right linear grammars.

One last definitional note. As a mechanism to allow the empty string as a member of a language we shall allow a production of the form

$$S \to \varepsilon$$

to appear in the rules for types 1 through 3 if the starting symbol $S$ never appears on the right hand side of any production. Thus $\varepsilon$ can be generated by a grammar but not used to destroy the length preserving nature of these grammars. A complete type 2 grammar for generating strings of the form $a^n b^n$ for $n \geq 0$ is:

$$S \to A$$
$$S \to \varepsilon$$
$$A \to aAb$$
$$A \to ab$$

As a recap of our definitions, here is a chart that summarizes the restrictions placed upon productions of all of the types of grammars. The strings $\alpha$ and $\beta$ are of the form $(N \cup T)^*$, $A$ and $B$ are nonterminals, and $d$ is a terminal symbol.

<table>
<thead>
<tr>
<th>Type</th>
<th>Production</th>
<th>Restriction</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>$\alpha \to \beta$</td>
<td>$</td>
<td>\alpha</td>
</tr>
<tr>
<td>1</td>
<td>$\alpha \to \beta$</td>
<td>$</td>
<td>\alpha</td>
</tr>
<tr>
<td>2</td>
<td>$A \to \beta$</td>
<td>$1 \geq</td>
<td>\beta</td>
</tr>
<tr>
<td>3</td>
<td>$A \to dB$</td>
<td></td>
<td>Regular or</td>
</tr>
<tr>
<td></td>
<td>$A \to d$</td>
<td></td>
<td>Right Linear</td>
</tr>
</tbody>
</table>
We shall close this section on grammars by presenting the grammar for a language which will later be shown not to be context free (type 2) like the previous examples. We need something a bit more complicated in order to show off our new string generating system. So, we shall design a context sensitive (type 1) grammar.

Let's try a grammar for strings of the form $0^n1^n0^n$. Here is the strategy. There are three main steps. They are:

a) Generate a string of the form $0^n(AB)^n$

b) Change this to one of the form $0^nA^nB^n$

c) Convert the A's to 1's and the B's to 0's

Thus our terminal set is $\{0,1\}$ and we have A and B as nonterminals so far. We shall introduce another nonterminal C to help coordinate the above steps and make sure that they take place in the proper order. Here are the details for the three steps above.

a) Generate an equal number of 0's, A's, and B's with the productions

$$S \rightarrow 0SAB$$
$$S \rightarrow 0CB$$

These productions generate strings such as:

$$000CBABAB.$$  
(By the way, the symbol C will eventually be like an A. For the moment, it will help change letters into zeros and ones.)

b) With the initial zeros are in place, we must group all of the A's and all of the B's together. This is done by the very context sensitive production:

$$BA \rightarrow AB$$

A derivation sequence of this might be:

$$000CBABAB$$
$$000CABBBAB$$
$$000CABABBB$$
c) Now it is time to change letters into numbers. The symbol C is used as a transformer. It moves to the right changing A's to ones until it reaches the first B. Thus, we need:

\[
\text{CA} \rightarrow 1C
\]

At this point we have generated:

\[
00011CBBB
\]

and can change all of the B's to zeros with:

\[
\text{CB} \rightarrow 10 \\
0B \rightarrow 00
\]

until we get the final string of terminal symbols we desire. A derivation sequence of this last part is:

\[
00011CBBB \\
0001110BB \\
00011100B \\
000111000
\]

Now let's collect the productions of the grammar we have designed.

\[
\begin{align*}
S & \rightarrow 0SAB & \text{CA} & \rightarrow 1C \\
S & \rightarrow 0CB & \text{CB} & \rightarrow 10 \\
BA & \rightarrow AB & 0B & \rightarrow 00
\end{align*}
\]

And here are three more example derivations of the same string:

\[
\begin{align*}
\text{S} & \rightarrow 0SAB & \text{S} & \rightarrow 0SAB & \text{S} & \rightarrow 0SAB \\
0SAB & \rightarrow 0SAB & 0SAB & \rightarrow 0SAB & 0SAB & \rightarrow 0SAB \\
00SABAB & \rightarrow 00SABAB & 00SABAB & \rightarrow 00SABAB \\
000CBABAB & \rightarrow 000CBABAB & 000CBABAB & \rightarrow 000CBABAB \\
000CABBAB & \rightarrow 000CABBAB & 000CABBAB & \rightarrow 000CABBAB \\
0001CBBAB & \rightarrow 0001CBBAB & 0001CBBAB & \rightarrow 0001CBBAB \\
0001CBAABB & \rightarrow 0001CBAABB & 0001CBAABB & \rightarrow 0001CBAABB \\
00001CBB & \rightarrow 00001CBB & 00001CBB & \rightarrow 00001CBB \\
0000011CBB & \rightarrow 0000111CBB & 0000111CBB & \rightarrow 0000111CBB \\
000001110BB & \rightarrow 00001110BB & 00001110BB & \rightarrow 00001110BB \\
0000011100B & \rightarrow 000011100B & 000011100B & \rightarrow 000011100B \\
00000111000 & \rightarrow 0000111000 & 0000111000 & \rightarrow 0000111000
\end{align*}
\]
The grammar presented above seems to generate strings of the appropriate form, but we really should back up and look at it again with the following questions in mind.

a) Are the same number of 0's, A's and B's generated?  
b) What happens if some B's change to 0's too early?  
c) Can all of the correct strings be generated?

It should be clear that our grammar does indeed generate strings of the form $0^n1^n0^n$. We should also note that the context sensitive nature of type 1 productions was very important in the design of our grammar. This was what allowed us to first arrange the A's and B's and then turn them into 1's and 0's.