First

First(ω) = { c | cs ∊ L(ω) where c is a terminal and s is a string of 0 or more terminals }

In words, First(ω) is the set of all first terminal symbols of all the strings derivable from ω. So, if we have the grammar

S → aSa | bSb | x

then First(aSa) = {a}, First(bSb) = {b}, First(x) = {x}, and First(S) = {a,b,x} because all strings derivable from aSa begin with 'a', all strings derivable from bSb begin with 'b', all strings derivable from x begin with 'x', and all strings derivable from S begin with 'a', 'b', or 'x'.

Deducing First:

First(λ) is { } because no string other than the empty string can be derived from the empty string, and the empty string has no first character.

First(a) is {a} because {a} is the set of strings derivable from a, and every string in that set begins with a.

First(aX) is {a} because every string derivable from aX begins with a, no matter what X is.

First(XY) when X *is not* nullable is the same as First(X) because every string in L(XY) begins with a string from L(X). (Note: X is "nullable" if the empty string can be derived from it. A nullable non-terminal can essentially be deleted during a derivation by letting it go to λ.)

First(XY) when X *is* nullable is First(X) ∪ First(Y) because every string in L(XY) begins with a string from L(X) and is followed by a string from L(Y). Since X is nullable there are derivations that begin XY → Y → ..., which means that the resulting string begins with a terminal from First(Y).

This last one is very important. The logic applies to any prefix of non-terminals that are nullable. If X and Y are both nullable, then First(XYZ) is equal to First(X) ∪ First(Y) ∪ First(Z) because X or XY could produce the empty string making a string from L(Y) or L(Z) lead the resulting derived string.

Patterns

Based on these observations, each production in a grammar can produce one or more pieces of information. Look for the following patterns and write down the information you can deduce. (In these patterns ω should be thought of as an arbitrary, possibly empty, string.)

A → λ produces no information about First sets because λ is length 0.

A → xω tells you x ∊ First(A) because a possible derivation beginning with A is A → xω → ... and any derivation beginning that way begins with x.

A → Bω tells you First(B) ⊆ First(A) because a possible derivation beginning with A is A → Bω → ... and any derivation beginning that way begins with a symbol from First(B).

A → Bω also tells you First(ω) ⊆ First(A) if B is nullable because a possible derivation beginning with A is A → Bω → ... → ω → ... and any derivation beginning that way begins with a symbol from First(ω).

You can often tell by just looking at a grammar what are all the First sets, but a methodical way to deduce them all is by fulfilling a system of set constraints. To do this, examine each production in a grammar and write down what it tells you as set statements as described above. For example...

Production	Indicates that
T → R	First(R) ⊆ First(T)
T → aTc	a ∈ First(T)
R → bR	b ∈ First(R)
R → λ	nothing about any First set since λ is length 0

Once you have these constraints, you next want to find the smallest sets that satisfy them. Here's an algorithm that does so.

1. Make a list of all non-terminals and initialize their First sets with the terminals required by the ∈ constraints.
2. For each ⊆ constraint, make it true by copying any missing elements from the set on the left-hand-side to the set on the right-hand-side.
3. If Step 2 changed anything, do Step 2 again.

Once Step 2 does not change anything, you have reached a "fixed point" in the algorithm and you should stop (because further iterations won't change anything either). Your final list of sets tells you the First set of each non-terminal. These can then be used to help determine the First set of each production's right-hand-side.

For example, after initialization First(T) = {a} and First(R) = {b}. After Step 2 First(T) = {a,b} and First(R) = {b}. After a second iteration of Step 2 First(T) = {a,b} and First(R) = {b} are still true, which means further iterations won't change anything and we're done.