1 Scanning

The goal of scanning is to take a non-empty input and split it into a non-empty sequence of tokens from some given language.

Formally, given a language $L$ (our set of tokens) and a word $w \neq \varepsilon$, we want to find words $w_1, \ldots, w_n \in L$, for some $n > 0$, such that $w = w_1 \cdots w_n$. The words $w_1, \ldots, w_n$ are the tokens, and concatenated together they form the original input $w$.

If it is possible to split a word into tokens like this, we say the word can be scanned with respect to $L$.

We discussed two scanning algorithms in class: Maximal Munch and Simplified Maximal Munch. These algorithms assume the set of tokens $L$ is a regular language and that we have a DFA recognizing $L$.

Both algorithms work by running the DFA with the input word $w$ until either all input is consumed, or the next transition leads to an error state. In either case, the following happens:

- **Simplified Maximal Munch** checks if the current state is accepting. If so, it outputs the portion of the input it consumed as a token, and then resets the DFA and continues consuming input (or stops if there is no more input). If not, it stops and produces an error indicating it failed to scan the input.

- **Maximal Munch** attempts to backtrack (in both the input and the DFA) to the last accepting state it passed through. If it is currently in an accepting state, this counts as the “last accepting state”. If it did not pass through any accepting states on the current run of the DFA, it stops and produces an error indicating it failed to scan the input. Otherwise, it outputs the portion of the input it consumed (accounting for backtracking) as a token, and then resets the DFA and continues consuming input (or stops if there is no more input).

It is possible that a string can be scanned, but Maximal Munch and Simplified Maximal Munch will fail to scan it. It is also possible that Maximal Munch will succeed in scanning a string but Simplified Maximal Munch will fail.
Exercise

Suppose our set of tokens for scanning is described by the following DFA:

![DFA Diagram](attachment:image.png)

Give the sequence of tokens produced by Simplified Maximal Munch for each input below. If there is an error, give the sequence of tokens produced before the error occurred, then write “ERROR”.

1. 0xa0xb0xcd
2. 0xend---
3. 1234−120xb
4. abcend--en−3
5. 01end−end10

For all of these inputs, it does not make a difference whether we use Maximal Munch or Simplified Maximal Munch. Can you find an input where Maximal Munch works but Simplified Maximal Munch produces an error?
2 LL(1) Parsing

An LL(1) parser is a *top-down* parser; it begins from the start symbol of the grammar and finds a derivation for the input string, as opposed to a *bottom-up parser* which starts from the input string and works backwards.

The 1 in LL(1) stands for “one symbol of lookahead”. This means the parser is only able to make decisions about which rule to apply based on the next symbol in the input, not any further symbols. As a result of this restriction, LL(1) parsing does not always work; there are some grammars and input words that cannot be parsed using this method.

To determine whether LL(1) parsing will work for a grammar, we must construct the *predict table* for the grammar. This is a table of production rules; if $A$ is a non-terminal and $a$ is a terminal, then $\text{Predict}(A, a)$ is the production rule we should apply when we see the nonterminal symbol $A$ in our derivation and see the terminal symbol $a$ in our input.

Informally, $\text{Predict}(A, a)$ contains a rule $A \rightarrow \gamma$ if:

- The string $\gamma$ can derive a string whose first symbol is $a$.
- The string $\gamma$ can derive $\varepsilon$, and it is possible for the terminal $a$ to immediately follow $A$ in a derivation.

The idea is that we want to apply a rule if it will lead us to something that starts with $a$. The second case is to account for the fact that in some cases, we might have to “get rid” of $A$ by deriving $\varepsilon$ first before getting to something that derives a string starting with $a$.

Here is the formal definition.

$$\text{Predict}(A, a) = \{ A \rightarrow \gamma \mid a \in \text{First}(\gamma) \text{ or } (\text{Nullable}(\gamma) \text{ and } a \in \text{Follow}(A)) \}$$

$$\text{First}(\gamma) = \{ b \mid \gamma \Rightarrow^* b\beta \text{ for some } \beta \}$$

$$\text{Follow}(A) = \{ c \mid S' \Rightarrow^* \alpha Ac\beta \text{ for some } \alpha, \beta \}$$

$$\text{Nullable}(\gamma) = (\text{true if } \gamma \Rightarrow^* \varepsilon, \text{false otherwise})$$

If $\text{Predict}(A, a)$ contains *at most one* rule for each pair $(A, a)$, we say that the grammar is LL(1), and we can use the LL(1) parsing algorithm with this grammar.

**Exercise**

Consider the following context-free grammar:

1. Compute the predict table for this grammar.

2. Use the predict table to perform a top-down parse of the string $\vdash appqb \dashv$ and draw the parse tree.