Constructing The Parsing Table

- Enter production $A \rightarrow \alpha$ into $ParsingTable[A, t]$ for all symbols $t \in Predict(A \rightarrow \alpha)$, where

$$Predict(A \rightarrow \alpha) = \begin{cases} (FIRST(\alpha) - \{\varepsilon\}) \cup FOLLOW(A), & \text{if } NULLABLE(\alpha) \\ FIRST(\alpha), & \text{otherwise} \end{cases}.$$

Constructing The Parsing Table

- Enter production $A \to \alpha$ into $ParsingTable[A, t]$ for all symbols $t \in Predict(A \to \alpha)$, where

$$Predict(A \to \alpha) = \begin{cases} (FIRST(\alpha) - \{\varepsilon\}) \cup FOLLOW(A), & \text{if } NULLABLE(\alpha) \\ FIRST(\alpha), & \text{otherwise} \end{cases}.$$

- Example

S→ A$

A → BC | x

B → t | ε

C → v | ε

NULLABLE = {A,B,C}
FIRST(A)={x,t,v,ε}
FIRST(B)={t,ε}
FIRST(C)={v,ε}
FIRST(S)={x,t,v,$}
FOLLOW(A)={$}
FOLLOW(B)={v,$}
FOLLOW(C)={$}

A → BC is in table entries for t,v,$
B → ε is in table entries for v,$
S→A$ is in table entries for x,t,v,$
C→ ε is in table entries for $

Parsing Tables for Non-LL(1) Grammars

- Multiple productions will map to the same entry of the parsing table, indicating a *conflict*, i.e., insufficient information to pick the next production with certainty.

Parsing Tables for Non-LL(1) Grammars

- Multiple productions will map to the same entry of the parsing table, indicating a *conflict*, i.e., insufficient information to pick the next production with certainty.
- Consider the grammar

  $S \rightarrow S{+}S\$ \mid S{*}S\$ \mid \textbf{num}\$$
- We find that

  $\text{FIRST}(S{+}S) = \text{FIRST}(S{*}S) = \text{FIRST}(\textbf{num}) = \{\textbf{num}\}$

	num	**+**	*****	**$**
S	$\rightarrow S{+}S$, $\rightarrow S{*}S$, $\rightarrow$ **num**			

More on Regular Grammars

- Regular grammars are a particularly simple form of context-free grammars in which all productions (before $\$$-augmentation) are of the form $A \rightarrow wB$ or $A \rightarrow w$, with $A, B \in N$ and $w \in \Sigma^*$.
  - This is also called a right-linear grammar.
  - Aside: There is an equivalent formalism with all productions being of the form $A \rightarrow Bw$ or $A \rightarrow w$, which is called a left-linear grammar.

More on Regular Grammars

- Regular grammars are a particularly simple form of context-free grammars in which all productions (before \$-augmentation) are of the form $A \to wB$ or $A \to w$, with $A, B \in N$ and $w \in \Sigma^*$.
  - This is also called a right-linear grammar.
  - Aside: There is an equivalent formalism with all productions being of the form $A \to Bw$ or $A \to w$, which is called a left-linear grammar.
- Consider the regular language specified by the regular expression $(a|b)^*abb$. The following is a right-linear grammar for this language.

$$
\begin{aligned}
&S \to T\$ \\
&T \to aA \mid bT \\
&A \to aA \mid bB \\
&B \to aA \mid bC \\
&C \to aA \mid bT \mid \$
\end{aligned}
$$

More on Regular Grammars

- Regular grammars are a particularly simple form of context-free grammars in which all productions (before \$-augmentation) are of the form $A \to wB$ or $A \to w$, with $A, B \in N$ and $w \in \Sigma^*$.
  - This is also called a right-linear grammar.
  - Aside: There is an equivalent formalism with all productions being of the form $A \to Bw$ or $A \to w$, which is called a left-linear grammar.
- Consider the regular language specified by the regular expression $(a|b)^*abb$. The following is a right-linear grammar for this language.

$$
\begin{aligned}
&S \to T\$ \\
&T \to aA \mid bT \\
&A \to aA \mid bB \\
&B \to aA \mid bC \\
&C \to aA \mid bT \mid \$
\end{aligned}
$$

- What does the "parsing table" of this grammar look like?
  - It is just the transition function of the DFA recognizing the language.

Towards A Table-Driven Predictive Parser

- Recall the table-driven DFA simulation strategy
  - Represent the transition function as an array `nextstate` indexed by the current state and ~~the class of~~ the character being scanned.
    `state = nextstate[state][nextchar];`

Towards A Table-Driven Predictive Parser

- Recall the table-driven DFA simulation strategy
  - Represent the transition function as an array `nextstate` indexed by the current state and ~~the class of~~ the character being scanned.
    `state = nextstate[state][nextchar];`
- We can't do exactly this for arbitrary CFLs, or even CFLs with an LL(1) grammar.
  - The issue is that the memory of the prefix examined and processed has to retain more information than just the path-oblivious (aka "Markovian") summary that the FA model allows.
  - This ultimately stems from the difference between the right-hand sides of productions (the unrestricted $A \to \alpha$ vs. the restricted $A \to wB$).
  - However, because the left-hand sides of productions are single non-terminals (i.e., context-free), this information can be organized in a LIFO manner, i.e., using a stack.

Towards A Table-Driven Predictive Parser

- Recall the table-driven DFA simulation strategy
  - Represent the transition function as an array `nextstate` indexed by the current state and ~~the class of~~ the character being scanned.
    `state = nextstate[state][nextchar];`
- We can't do exactly this for arbitrary CFLs, or even CFLs with an LL(1) grammar.
  - The issue is that the memory of the prefix examined and processed has to retain more information than just the path-oblivious (aka "Markovian") summary that the FA model allows.
  - This ultimately stems from the difference between the right-hand sides of productions (the unrestricted $A \rightarrow \alpha$ vs. the restricted $A \rightarrow wB$).
  - However, because the left-hand sides of productions are single non-terminals (i.e., context-free), this information can be organized in a LIFO manner, i.e., using a stack.
- So our table-driven predictive parser will work using a parsing stack, in addition to the input tape and the parsing table.

A Table-Driven Predictive Parser

- Initialize the parsing stack to [$,S], with S being at TOS.
- At each step, act based on the TOS symbol X and the input token a.
  - If $X = a = \$$, **halt**; the parse is successful.
  - If $X = a \neq \$$, pop the stack and advance the input to the next token.
  - Otherwise, $X \in N$.
    - If $ParsingTable[X, a] = X \rightarrow Y_1 \cdots Y_k$:
      - Pop the stack.
      - $Push(Y_k). \cdots Push(Y_1)$.
    - If $ParsingTable[X, a]$ is empty, there is an **error** in the input.

A Table-Driven Predictive Parser

- Initialize the parsing stack to [$,S], with S being at TOS.
- At each step, act based on the TOS symbol X and the input token a.
  - If $X = a = \$$, **halt**; the parse is successful.
  - If $X = a \neq \$$, pop the stack and advance the input to the next token.
  - Otherwise, $X \in N$.
    - If $ParsingTable[X, a] = X \rightarrow Y_1 \cdots Y_k$:
      - Pop the stack.
      - $Push(Y_k). \cdots Push(Y_1)$.
    - If $ParsingTable[X, a]$ is empty, there is an **error** in the input.

- Verify that this devolves to the table-driven DFA simulation if the grammar is regular.