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:

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. \textbf{0xa0xb0xcd}

\textbf{Solution:} Simplified Maximal Munch works by moving through the DFA until either there is no transition on the next symbol, or we reach the end of input. Then we either output a token or produce an error, depending on if we are in an accepting state. This leads to the following sequence of tokens:

\textbf{HEXINT 0xa0}
\textbf{ID xb0xcd}

This is may be somewhat unexpected, since clearly another way to scan the string is three \textbf{HEXINT} tokens in a row. But this is what Simplified Maximal Munch will produce because it always takes the longest token possible.
2. 0xend---
   Solution: Again we have to be careful and simply tracing through the DFA, rather than looking at a logical way to break the word into tokens. We get:
   
   \[
   \text{HEXINT 0xe}
   \]
   \[
   \text{ID nd}
   \]
   \[
   \text{DECR --}
   \]
   \[
   \text{MINUS -}
   \]
   
   Note that the dash in the groups of transitions like “a-d, f-z” from START to ID is to indicate a range; it is not a minus sign.

3. 1234-120xb
   Solution:
   
   \[
   \text{INT 1234}
   \]
   \[
   \text{INT -120}
   \]
   \[
   \text{ID xb}
   \]

4. abcend--en-3
   This is an example where we run into an error. Solution:
   
   \[
   \text{ID abcend}
   \]
   \[
   \text{DECR --}
   \]
   \[
   \text{ERROR}
   \]
   
   After reading these two tokens, we read e and go to state E, then n and go to state EN. Then we read a minus sign – and there is no transition out of this state on –. Since we are not in an accepting state, this is an error.

   In fact Maximal Munch would also produce an error here. In Maximal Munch we would backtrack to the last accepting state, but we did not pass through any accepting states on the way to EN.

5. 01end-end10
   Solution:
   
   \[
   \text{ZERO 0}
   \]
   \[
   \text{INT 1}
   \]
   \[
   \text{END end}
   \]
   \[
   \text{MINUS -}
   \]
   \[
   \text{ID end10}
   \]

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?

Solution: An example is 0x. Simplified Maximal Munch will reach the state ZEROX, then since it is the end of input, it produces an error since the state is not accepting. Maximal Munch, however, will backtrack to the last accepting state ZERO and output a ZERO 0 token. Then Maximal Munch continues and outputs an ID x token.
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 sets of production rules; if $A$ is a non-terminal and $a$ is a terminal, then $\text{Predict}(A,a)$ is the set of possible production rules we could 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.

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

- $\text{First}(\gamma) = \{ b | \ \gamma \Rightarrow ^* b\beta \text{ for some } \beta \}$
- $\text{Follow}(A) = \{ c | S' \Rightarrow ^* \alpha Ac\beta \text{ for some } \alpha, \beta \}$
- $\text{Nullable}(\gamma) = \text{true if } \gamma \Rightarrow ^* \varepsilon, \text{ false otherwise}$

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.

Solution: When computing a predict table, you want to do things in the following order: Nullable, First, then Follow. You can either use the algorithms shown in class or just use intuition based on the definitions. We will be using a more intuitive approach.

I prefer to fill out the predict table as I compute Nullable, First and Follow, rather than computing all three and then filling out the predict table at the end.
First let’s compute Nullable(A) for each non-terminal A. In addition to this, for each non-terminal A we will keep track of which rules can be used to nullify A (that is, derive ε from A). These are called the nullable rules for A. Keeping track of these rules will help with filling out the predict table.

- Nullable(X) and Nullable(Y) are clearly true since they directly derive ε. Rule (5) is a nullable rule for X and rule (7) is a nullable rule for Y.
- Nullable(S) is true because we can apply rule (3), S → XY, and then nullify X and Y. Rule (3) is a nullable rule for X.
- Nullable(S’) is false because the only rule we can expand S’ with is rule (1), and it has a terminal on the right hand side.

We end up with the following data:

<table>
<thead>
<tr>
<th>Non-terminal</th>
<th>Nullable</th>
<th>Nullable rules</th>
</tr>
</thead>
<tbody>
<tr>
<td>S’</td>
<td>false</td>
<td>none</td>
</tr>
<tr>
<td>S</td>
<td>true</td>
<td>(3)</td>
</tr>
<tr>
<td>X</td>
<td>true</td>
<td>(5)</td>
</tr>
<tr>
<td>Y</td>
<td>true</td>
<td>(7)</td>
</tr>
</tbody>
</table>

Next let’s compute First(A) for each non-terminal A. The idea is to look at the first symbol on the right-hand-side of each rule that expands A.

We will also start filling in the predict table. Each time we add a letter a to First(A), we update Predict(A, a) to include the rule we used.

- S’ is only expanded by rule (1), S’ → S ⊢. Thus we have First(S’) = {⊢}. We add rule (1) to Predict(S’, ⊢).
- X is expanded by rules (4) and (5).
  - Rule (4), X → pX, tells us that p ∈ First(X).
  - Rule (5), X → ε, can be ignored since there are no symbols on the right hand side.
Thus First(X) = {p}. We add rule (4) to Predict(X, p).
- Y is expanded by rules (6) and (7).
  - Rule (6), Y → q, tells us that q ∈ First(Y).
  - Rule (7), Y → ε, is ignored.
Thus First(Y) = {q}. We add rule (6) to Predict(Y, q).
- S is expanded by rules (2) and (3).
  - Rule (2), S → aXYb, tells us that a ∈ First(S).
  - Rule (3), S → XY, is a little tricky to deal with. First, notice that anything that is in First(X) is also in First(S), since the right hand side of S → XY starts with X. But also, since X is nullable, we also get everything that’s in First(Y). So this rule tells us that p, q ∈ First(S).
Thus First(S) = {a, p, q}. We add rule (2) to Predict(S, a), and we add rule (3) to both Predict(S, p) and Predict(S, q).
Here are the resulting First sets:

<table>
<thead>
<tr>
<th>Non-terminal</th>
<th>First</th>
</tr>
</thead>
<tbody>
<tr>
<td>S′</td>
<td>{⊣}</td>
</tr>
<tr>
<td>S</td>
<td>{a, p, q}</td>
</tr>
<tr>
<td>X</td>
<td>{p}</td>
</tr>
<tr>
<td>Y</td>
<td>{q}</td>
</tr>
</tbody>
</table>

And here is our partially-filled predict table:

<table>
<thead>
<tr>
<th>Predict</th>
<th>⊢</th>
<th>⊣</th>
<th>a</th>
<th>b</th>
<th>p</th>
<th>q</th>
</tr>
</thead>
<tbody>
<tr>
<td>S′</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>S</td>
<td></td>
<td>2</td>
<td>3</td>
<td>3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>X</td>
<td></td>
<td></td>
<td>4</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Y</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>6</td>
</tr>
</tbody>
</table>

Finally, we do Follow sets. These are the most difficult to compute. To compute Follow(A), we look at rules in which A appears on the right hand side, and try to figure out what symbols can possibly appear after A in a derivation.

Updating the predict table here is a bit tricky. Similarly to First sets, whenever we add a letter a to Follow(A), we update Predict(A, a). However, the way we update it is different: we add all nullable rules for A to Predict(A, a). The idea behind this is, if we want to match a letter a while parsing, and we have an A as the leftmost term in our derivation with an a following it, we should apply a rule that gets rid of the A. This is precisely a nullable rule for A.

- Follow(S′) = ∅ because S′ does not appear on the right hand side of any rule.
- Follow(S) = {⊣} because S appears on the right hand side of S′ →⊣ S ⊣ and is followed by the terminal ⊣. We add the nullable rule (3) for S to Predict(S, ⊣).
- Follow(Y) is harder. First, because Y appears on the right hand side of S → aXYb and is followed by b, we add b to Follow(Y).
  
  But also, Y appears on the right hand side of S → XY, and it is at the end of the rule. This means that anything which can follow S can also follow Y.

  So we get Follow(Y) = {b} ∪ Follow(S) = {b, ⊣}. We add the nullable rule (7) to Predict(Y, b) and Predict(Y, ⊣).

- Follow(X) is even more complicated. It appears on the right hand side of S → aXYb, and is followed by Y. This means that anything in First(Y) can follow X.
  
  But also, Y is nullable, so b can follow X as well.

  Additionally we have the rule S → XY. Since Y is nullable, anything that can follow S can follow X for the same reasoning as Y.

  We also have X in the rule X → pX but this doesn’t tell us anything about what can follow X.

  Putting everything together we get Follow(X) = First(Y) ∪ {b} ∪ Follow(S) = {b, q, ⊣}. We add nullable rule (5) to Predict(X, b), Predict(X, q) and Predict(X, ⊣).

Here are the resulting Follow sets:

<table>
<thead>
<tr>
<th>Non-terminal</th>
<th>Follow</th>
</tr>
</thead>
<tbody>
<tr>
<td>S′</td>
<td>∅</td>
</tr>
<tr>
<td>S</td>
<td>{⊣}</td>
</tr>
<tr>
<td>X</td>
<td>{b, q, ⊣}</td>
</tr>
<tr>
<td>Y</td>
<td>{b, ⊣}</td>
</tr>
</tbody>
</table>
And finally, here is our complete predict table:

<table>
<thead>
<tr>
<th>Predict</th>
<th>⊢</th>
<th>a</th>
<th>b</th>
<th>p</th>
<th>q</th>
</tr>
</thead>
<tbody>
<tr>
<td>S'</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>S</td>
<td>3</td>
<td>2</td>
<td>3</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>5</td>
<td>5</td>
<td>4</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>Y</td>
<td>7</td>
<td>7</td>
<td>6</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

What if we didn’t fill out the predict table as we computed Nullable, First and Follow? We can still build it up with the following method. For each rule $A \rightarrow \gamma$ in the grammar:

- Compute First($\gamma$). You can use the First sets for the non-terminals to help with this. For each $a \in$ First($\gamma$), add the rule $A \rightarrow \gamma$ to $\text{Predict}(A,a)$.

- Compute Nullable($\gamma$). You can use the Nullable values for the non-terminals to help with this. If Nullable($\gamma$) is true, then for each $a \in$ Follow($A$), add the rule $A \rightarrow \gamma$ to $\text{Predict}(A,a)$.

Here is an example of computing the predict table this way.

- For $S' \rightarrow \vdash S \dashv$ (1), we have First($\vdash S \dashv$) = {$\dashv$} and $\vdash S \dashv$ is not nullable. Add (1) to $\text{Predict}(S', \dashv)$.

- For $S \rightarrow aXYb$ (2), we have First($aXYb$) = {$a$} and $aXYb$ is not nullable. Add (2) to $\text{Predict}(S, a)$.

- For $S \rightarrow XY$ (3), we have First($XY$) = First($X$) $\cup$ First($Y$) = {$p, q$} since $X$ is nullable. Add (3) to $\text{Predict}(S, p)$ and $\text{Predict}(S, q)$. Additionally, $XY$ is nullable and Follow($S$) = {$\dashv$}, so add (3) to $\text{Predict}(S, \dashv)$.

- For $X \rightarrow pX$ (4), we have First($pX$) = {$p$} and $pX$ is not nullable. Add (4) to $\text{Predict}(X, p)$.

- For $X \rightarrow \varepsilon$ (5), First($\varepsilon$) is empty. However $\varepsilon$ is trivially nullable and Follow($X$) = {$b, q, \dashv$}, so add (5) to $\text{Predict}(X, b)$, $\text{Predict}(X, q)$ and $\text{Predict}(X, \dashv)$.

- For $Y \rightarrow q$ (6), we have First($q$) = {$q$} and $q$ is not nullable. Add (6) to $\text{Predict}(Y, q)$.

- For $Y \rightarrow \varepsilon$ (7), First($\varepsilon$) is empty. However $\varepsilon$ is trivially nullable and Follow($Y$) = {$b, \dashv$}, so add (y) to $\text{Predict}(Y, b)$ and $\text{Predict}(Y, \dashv)$.

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

**Solution:** The table on the next page shows a trace of the algorithm. In the table, the top of the stack is on the right, and the bottom is on the left. Note that when applying a rule, we push the right-hand-side onto the stack in reverse order, so that the top has the leftmost symbol of the rule.
<table>
<thead>
<tr>
<th>Stack</th>
<th>Consumed Input</th>
<th>Remaining Input</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>$S'$</td>
<td>$\varepsilon$</td>
<td>$\vdash appqb \vdash$</td>
<td>Apply rule (1): Pop $S'$, push $\vdash$, $S$, $\vdash$</td>
</tr>
<tr>
<td>$\vdash S \vdash$</td>
<td>$\varepsilon$</td>
<td>$\vdash appqb \vdash$</td>
<td>Match $\vdash$: Pop $\vdash$, consume $\vdash$ from input</td>
</tr>
<tr>
<td>$\vdash S$</td>
<td>$\vdash$</td>
<td>$appqb \vdash$</td>
<td>Apply rule (2): Pop $S$, push $b$, $Y$, $X$, $a$</td>
</tr>
<tr>
<td>$\vdash bYXa$</td>
<td>$\vdash$</td>
<td>$appqb \vdash$</td>
<td>Match $a$: Pop $a$, consume $a$ from input</td>
</tr>
<tr>
<td>$\vdash bYX$</td>
<td>$\vdash a$</td>
<td>$ppqb \vdash$</td>
<td>Apply rule (4): Pop $X$, push $X$, $p$</td>
</tr>
<tr>
<td>$\vdash bYXp$</td>
<td>$\vdash a$</td>
<td>$ppqb \vdash$</td>
<td>Match $p$: Pop $p$, consume $p$ from input</td>
</tr>
<tr>
<td>$\vdash bYX$</td>
<td>$\vdash ap$</td>
<td>$pqb \vdash$</td>
<td>Apply rule (4): Pop $X$, push $X$, $p$</td>
</tr>
<tr>
<td>$\vdash bYXp$</td>
<td>$\vdash ap$</td>
<td>$pqb \vdash$</td>
<td>Match $p$: Pop $p$, consume $p$ from input</td>
</tr>
<tr>
<td>$\vdash bYX$</td>
<td>$\vdash app$</td>
<td>$qb \vdash$</td>
<td>Apply rule (5): Pop $X$ (don’t push anything)</td>
</tr>
<tr>
<td>$\vdash bY$</td>
<td>$\vdash app$</td>
<td>$qb \vdash$</td>
<td>Apply rule (6): Pop $Y$, push $q$</td>
</tr>
<tr>
<td>$\vdash bq$</td>
<td>$\vdash app$</td>
<td>$qb \vdash$</td>
<td>Match $q$: Pop $q$, consume $q$ from input</td>
</tr>
<tr>
<td>$\vdash b$</td>
<td>$\vdash appq$</td>
<td>$b \vdash$</td>
<td>Match $b$: Pop $b$, consume $b$ from input</td>
</tr>
<tr>
<td>$\vdash$</td>
<td>$\vdash appqb \vdash$</td>
<td>$\varepsilon$</td>
<td>Match $\vdash$: Pop $\vdash$, consume $\vdash$ from input</td>
</tr>
<tr>
<td>$\varepsilon$</td>
<td>$\vdash appqb \vdash$</td>
<td>$\varepsilon$</td>
<td>Accept: stack and remaining input are both empty</td>
</tr>
</tbody>
</table>

Reading the production rules that were applied from top to bottom gives a leftmost derivation. We can use this derivation to obtain the parse tree.

![Parse Tree](image-url)