For \( A \rightarrow X_1...X_n \), \( \text{Predict}(A \rightarrow X_1...X_n) = \text{Set of all initial (first) tokens derivable from } A \rightarrow X_1...X_n \)
\( = \{ a \in V_t \mid A \rightarrow X_1...X_n \Rightarrow^* a... \} \)
For example, given
\[
\text{Stmt} \rightarrow \text{Label id } = \text{Expr} ; \\
| \text{Label if Expr then Stmt} ; \\
| \text{Label read ( IdList )} ; \\
| \text{Label id ( Args )} ;
\]
\( \text{Label} \rightarrow \text{intlit} : \\
| \lambda \)

<table>
<thead>
<tr>
<th>Production</th>
<th>Predict Set</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \text{Stmt} \rightarrow \text{Label id } = \text{Expr} ; )</td>
<td>{id, intlit}</td>
</tr>
<tr>
<td>( \text{Stmt} \rightarrow \text{Label if Expr then Stmt} ; )</td>
<td>{if, intlit}</td>
</tr>
<tr>
<td>( \text{Stmt} \rightarrow \text{Label read ( IdList )} ; )</td>
<td>{read, intlit}</td>
</tr>
<tr>
<td>( \text{Stmt} \rightarrow \text{Label id ( Args )} ; )</td>
<td>{id, intlit}</td>
</tr>
</tbody>
</table>

Now let's parse
\( \text{id ( intlit ) ;} \)
Our start symbol is \( \text{Stmt} \) and the initial token is \( \text{id} \).
\( \text{id} \) can predict
\( \text{Stmt} \rightarrow \text{Label id } = \text{Expr} ; \)
\( \text{id} \) then predicts \( \text{Label} \rightarrow \lambda \)
The \( \text{id} \) is matched, but “(“ doesn’t match “=” so we backup and try a different production for \( \text{Stmt} \).
\( \text{id} \) also predicts
\( \text{Stmt} \rightarrow \text{Label id ( Args )} ; \)
Again, \( \text{Label} \rightarrow \lambda \) is predicted and used, and the input tokens can match the rest of the remaining production.
We had only one misprediction, which is better than before.
Now we'll rewrite the productions a bit to make predictions easier.

We now will match a production \( p \) only if the next unmatched token is in \( p \)'s predict set. We'll avoid trying productions that clearly won't work, so parsing will be faster.
But what is the predict set of a \( \lambda \)-production?
It can't be what's generated by \( \lambda \) (which is nothing!), so we'll define it as the tokens that can follow the use of a \( \lambda \)-production.
That is, \( \text{Predict}(A \rightarrow \lambda) = \text{Follow}(A) \)
where (by definition)
\( \text{Follow}(A) = \{ a \in V_t \mid S \Rightarrow^* Aa... \} \)
In our example,
\( \text{Follow}(\text{Label} \rightarrow \lambda) = \{ \text{id, if, read} \} \)
(since these terminals can immediately follow uses of Label in the given productions).

We remove the \( \text{Label} \) prefix from all the statement productions (now \( \text{intlit} \) won't predict all four productions).
We now have
\( \text{Stmt} \rightarrow \text{Label BasicStmt} \)
\( \text{BasicStmt} \rightarrow \text{id } = \text{Expr} ; \\
| \text{if Expr then Stmt} ; \\
| \text{read ( IdList )} ; \\
| \text{id ( Args )} ;
\)
\( \text{Label} \rightarrow \text{intlit} : \\
| \lambda \)
Now \( \text{id} \) predicts two different \( \text{BasicStmt} \) productions. If we rewrite these two productions into
\( \text{BasicStmt} \rightarrow \text{id } \text{StmtSuffix} \)
\( \text{StmtSuffix} \rightarrow = \text{Expr} ; \\
| ( \text{Args} ) ; \)
we no longer have any doubt over which production id predicts. We now have

<table>
<thead>
<tr>
<th>Production</th>
<th>Predict Set</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stmt → Label BasicStmt</td>
<td>Not needed!</td>
</tr>
<tr>
<td>BasicStmt → id StmtSuffix</td>
<td>(id)</td>
</tr>
<tr>
<td>BasicStmt → if Expr then Stmt ;</td>
<td>(if)</td>
</tr>
<tr>
<td>BasicStmt → read ( IdList ) ;</td>
<td>(read)</td>
</tr>
<tr>
<td>StmtSuffix → ( Args ) ;</td>
<td>{}</td>
</tr>
<tr>
<td>StmtSuffix → = Expr ;</td>
<td>{}</td>
</tr>
<tr>
<td>Label → intlit :</td>
<td>(intlit)</td>
</tr>
<tr>
<td>Label → λ</td>
<td>(if, id, read)</td>
</tr>
</tbody>
</table>

This grammar generates the same statements as our original grammar did, but now prediction never fails!

Whenever we must decide what production to use, the predict sets for productions with the same lefthand side are always disjoint. Any input token will predict a unique production or no production at all (indicating a syntax error).
If we never mispredict a production, we never backup, so parsing will be fast and absolutely accurate!

**LL(1) Grammars**

A context-free grammar whose predict sets are always disjoint (for the same non-terminal) is said to be LL(1).

LL(1) grammars are ideally suited for top-down parsing because it is always possible to correctly predict the expansion of any non-terminal. No backup is ever needed.

Formally, let
First(X₁...Xₙ) =  
{a in Vₜ | A → X₁...Xₙ ⇒⁺ a...}
Follow(A) = {a in Vₜ | S ⇒⁺ ...Aa...}

Predict(A → X₁...Xₙ) =
If X₁...Xₙ ⇒⁺ ☐
Then First(X₁...Xₙ) U Follow(A)
Else First(X₁...Xₙ)

If some CFG, G, has the property that for all pairs of distinct productions with the same lefthand side, A → X₁...Xₙ and A → Y₁...Yₘ it is the case that
Predict(A → X₁...Xₙ) ∩ Predict(A → Y₁...Yₘ) = ☐
then G is LL(1).

LL(1) grammars are easy to parse in a top-down manner since predictions are always correct.
Since the predict sets of both B productions and both D productions are disjoint, this grammar is LL(1).

<table>
<thead>
<tr>
<th>Production</th>
<th>Predict Set</th>
</tr>
</thead>
<tbody>
<tr>
<td>S → A a</td>
<td>{b,d,a}</td>
</tr>
<tr>
<td>A → B D</td>
<td>{b, d, a}</td>
</tr>
<tr>
<td>B → b</td>
<td>{b}</td>
</tr>
<tr>
<td>B → λ</td>
<td>{d, a}</td>
</tr>
<tr>
<td>D → d</td>
<td>{d}</td>
</tr>
<tr>
<td>D → λ</td>
<td>{a}</td>
</tr>
</tbody>
</table>

This approach is called recursive descent because the parsing procedures were typically recursive, and they descended down the input's parse tree (as top-down parsers always do).

**Building A Recursive Descent Parser**

We start with a procedure `Match`, that matches the current input token against a predicted token:

```c
void Match(Terminal a) {
    if (a == currentToken)
        currentToken = Scanner();
    else SyntaxError();
}
```

To build a parsing procedure for a non-terminal A, we look at all productions with A on the lefthand side:

\[ A \rightarrow X_1...X_n | A \rightarrow Y_1...Y_m | ... \]

We use predict sets to decide which production to match (LL(1) grammars always have disjoint predict sets).

We match a production's righthand side by calling `Match` to
match terminals, and calling parsing procedures to match non-terminals.

The general form of a parsing procedure for
\[ A \rightarrow X_1 \ldots X_n \mid A \rightarrow Y_1 \ldots Y_m \mid \ldots \]
is

```c
void A() {
    if (currentToken in Predict(A→X_1...X_n))
        for(i=1;i<=n;i++)
            if (X[i] is a terminal)
                Match(X[i]);
            else X[i]();
    else
        if (currentToken in Predict(A→Y_1...Y_m))
            for(i=1;i<=m;i++)
                if (Y[i] is a terminal)
                    Match(Y[i]);
                else Y[i]();
        else // Handle other A →... productions
            else // No production predicted
                SyntaxError();
}
```

Usually this general form isn’t used.
Instead, each production is “macro-expanded” into a sequence of `Match` and parsing procedure calls.

**Example: CSX-Lite**

<table>
<thead>
<tr>
<th>Production</th>
<th>Predict Set</th>
</tr>
</thead>
<tbody>
<tr>
<td>Prog → {Stmts} Eof</td>
<td>{</td>
</tr>
<tr>
<td>Stmts → Stmt Stmts</td>
<td>id if</td>
</tr>
<tr>
<td>Stmt → id = Expr ;</td>
<td>id</td>
</tr>
<tr>
<td>Stmt → if (Expr) Stmt</td>
<td>if</td>
</tr>
<tr>
<td>Expr → id Etail</td>
<td>id</td>
</tr>
<tr>
<td>Etail → + Expr</td>
<td>+</td>
</tr>
<tr>
<td>Etail → - Expr</td>
<td>-</td>
</tr>
<tr>
<td>Etail → λ</td>
<td>) ;</td>
</tr>
</tbody>
</table>

**CSX-Lite Parsing Procedures**

```c
void Prog() {
    Match("*");
    Stmts();
    Match("*");
    Match(Eof);
}

void Stmts() {
    if (currentToken == id ||
        currentToken == if){
        Stmt();
        Stmts();
    } else {
        /* null */
    }
}

void Stmt() {
    if (currentToken == id){
        Match(id);
        Match("=");
        Expr();
        Match(";");
    } else {
        Match(if);
        Match("=");
        Expr();
        Match(";");
        Stmts();
    }
}
```
void Expr() {
    Match(id);
    Etail();
}

void Etail() {
    if (currentToken == "+") {
        Match("+");
        Expr();
    } else if (currentToken == "-") {
        Match("-");
        Expr();
    } else {
        /* null */
    }
}

Let's use recursive descent to parse
{ a = b + c; } Eof
We start by calling Prog() since this represents the start symbol.

<table>
<thead>
<tr>
<th>Calls Pending</th>
<th>Remaining Input</th>
</tr>
</thead>
<tbody>
<tr>
<td>Match(&quot;=&quot;); Expr(); Match(&quot;;&quot;); Stmts(); Match(&quot;;&quot;); Match(Eof);</td>
<td>= b + c; } Eof</td>
</tr>
<tr>
<td>Expr(); Match(&quot;;&quot;); Stmts(); Match(&quot;;&quot;); Match(Eof);</td>
<td>b + c; } Eof</td>
</tr>
<tr>
<td>Match(id); Etail(); Stmts(); Match(&quot;;&quot;); Match(Eof);</td>
<td>b + c; } Eof</td>
</tr>
<tr>
<td>Etail(); Match(&quot;;&quot;); Stmts(); Match(&quot;;&quot;); Match(Eof);</td>
<td>+ c; } Eof</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Calls Pending</th>
<th>Remaining Input</th>
</tr>
</thead>
<tbody>
<tr>
<td>Match(&quot;=&quot;); Expr(); Match(&quot;;&quot;); Stmts(); Match(&quot;;&quot;); Match(Eof);</td>
<td>+ c; } Eof</td>
</tr>
<tr>
<td>Expr(); Match(&quot;;&quot;); Stmts(); Match(&quot;;&quot;); Match(Eof);</td>
<td>c; } Eof</td>
</tr>
<tr>
<td>Match(id); Etail(); Match(&quot;;&quot;); Stmts(); Match(&quot;;&quot;); Match(Eof);</td>
<td>c; } Eof</td>
</tr>
<tr>
<td>Etail(); Match(&quot;;&quot;); Stmts(); Match(&quot;;&quot;); Match(Eof);</td>
<td>; } Eof</td>
</tr>
<tr>
<td>/* null */ Match(&quot;;&quot;); Stmts(); Match(&quot;;&quot;); Match(Eof);</td>
<td>; } Eof</td>
</tr>
</tbody>
</table>
Syntax Errors in Recursive Descent Parsing

In recursive descent parsing, syntax errors are automatically detected. In fact, they are detected as soon as possible (as soon as the first illegal token is seen).

How? When an illegal token is seen by the parser, either it fails to predict any valid production or it fails to match an expected token in a call to `Match`.

Let's see how the following illegal CSX-lite program is parsed:

```
{ b + c = a; } Eof
```
(Where should the first syntax error be detected?)
**Table-Driven Top-Down Parsers**

Recursive descent parsers have many attractive features. They are actual pieces of code that can be read by programmers and extended. This makes it fairly easy to understand how parsing is done. Parsing procedures are also convenient places to add code to build ASTs, or to do type-checking, or to generate code. A major drawback of recursive descent is that it is quite inconvenient to change the grammar being parsed. Any change, even a minor one, may force parsing procedures to be reprogrammed, as productions and predict sets are modified. To a less extent, recursive descent parsing is less efficient than it might be, since subprograms are called just to match a single token or to recognize a righthand side.

An alternative to parsing procedures is to encode all prediction in a parsing table. A pre-programed driver program can use a parse table (and list of productions) to parse any LL(1) grammar. If a grammar is changed, the parse table and list of productions will change, but the driver need not be changed.

#### LL(1) Parse Tables

An LL(1) parse table, $T$, is a two-dimensional array. Entries in $T$ are production numbers or blank (error) entries. $T$ is indexed by:

- $A$, a non-terminal. $A$ is the non-terminal we want to expand.
- $CT$, the current token that is to be matched.
- $T[A][CT] = A \rightarrow X_1...X_n$ if $CT$ is in Predict($A \rightarrow X_1...X_n$)
- $T[A][CT] = \text{error}$ if $CT$ predicts no production with $A$ as its lefthand side.

#### CSX-lite Example

<table>
<thead>
<tr>
<th>Production</th>
<th>Predict Set</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 Prog $\rightarrow$ (Stmts) Eof</td>
<td>(</td>
</tr>
<tr>
<td>2 Stmts $\rightarrow$ Stmt Stmts</td>
<td>id if</td>
</tr>
<tr>
<td>3 Stmts $\rightarrow$ $\lambda$</td>
<td>)</td>
</tr>
<tr>
<td>4 Stmt $\rightarrow$ id $=$ Expr ;</td>
<td>id</td>
</tr>
<tr>
<td>5 Stmt $\rightarrow$ if (Expr) Stmt</td>
<td>if</td>
</tr>
<tr>
<td>6 Expr $\rightarrow$ id Etail</td>
<td>id</td>
</tr>
<tr>
<td>7 Etail $\rightarrow$ + Expr</td>
<td>+</td>
</tr>
<tr>
<td>8 Etail $\rightarrow$ - Expr</td>
<td>-</td>
</tr>
<tr>
<td>9 Etail $\rightarrow$ $\lambda$</td>
<td>) ;</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>( )</th>
<th>if ( )</th>
<th>id</th>
<th>=</th>
<th>+</th>
<th>-</th>
<th>;</th>
<th>eof</th>
</tr>
</thead>
<tbody>
<tr>
<td>Prog</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Stmts</td>
<td>3</td>
<td>2</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Stmt</td>
<td>5</td>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Expr</td>
<td></td>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Etail</td>
<td>9</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>