As an example consider
\[
S \rightarrow A \ B \ C \\
A \rightarrow a \\
B \rightarrow C \ D \\
D \rightarrow d \\
| \lambda \\
C \rightarrow c \\
| \lambda
\]

Recall that compilers prefer an unambiguous grammar because a unique parse tree structure can be guaranteed for all inputs. Hence a unique translation, guided by the parse tree structure, will be obtained.

We would like an algorithm that checks if a grammar is ambiguous.

Unfortunately, it is undecidable whether a given CFG is ambiguous, so such an algorithm is impossible to create.

Fortunately for certain grammar classes, including those for which we can generate parsers, we can prove included grammars are unambiguous.

Potentially, the most serious flaw that a grammar might have is that it generates the “wrong language.”

This is a subtle point as a grammar serves as the definition of a language.

For established languages (like C or Java) there is usually a suite of programs created to test and validate new compilers. An incorrect grammar will almost certainly lead to incorrect compilations of test programs, which can be automatically recognized.

For new languages, initial implementors must thoroughly test the parser to verify that inputs are scanned and parsed as expected.

**Parsers and Recognizers**

Given a sequence of tokens, we can ask:

"Is this input syntactically valid?"

(Is it generable from the grammar?).

A program that answers this question is a recognizer.

Alternatively, we can ask:

"Is this input valid and, if it is, what is its structure (parse tree)?"

A program that answers this more general question is termed a parser.

We plan to use language structure to drive compilers, so we will be especially interested in parsers.
Two general approaches to parsing exist.
The first approach is top-down.
A parser is top-down if it "discovers" the parse tree corresponding to a token sequence by starting at the top of the tree (the start symbol), and then expanding the tree (via predictions) in a depth-first manner.
Top-down parsing techniques are predictive in nature because they always predict the production that is to be matched before matching actually begins.

Consider
\[
E \rightarrow E + T \mid T \\
T \rightarrow T * id \mid id
\]
To parse \(id + id\) in a top-down manner, a parser build a parse tree in the following steps:

\[
\begin{align*}
E & \rightarrow E & \rightarrow E & \rightarrow E \\
& \rightarrow E + T & \rightarrow E + T & \rightarrow T \\
& \rightarrow E & \rightarrow E + T & \\
& \rightarrow T & \rightarrow T & \\
& \rightarrow id & \rightarrow id & id
\end{align*}
\]

A wide variety of parsing techniques take a different approach.
They belong to the class of bottom-up parsers.
As the name suggests, bottom-up parsers discover the structure of a parse tree by beginning at its bottom (at the leaves of the tree which are terminal symbols) and determining the productions used to generate the leaves.
Then the productions used to generate the immediate parents of the leaves are discovered.
The parser continues until it reaches the production used to expand the start symbol.
At this point the entire parse tree has been determined.

A bottom-up parse of \(id_1 + id_2\) would follow the following steps:

\[
\begin{align*}
T & \Rightarrow E & \Rightarrow E \\
& \Rightarrow T & \Rightarrow E \\
& \Rightarrow id_1 & \Rightarrow id_1 \\
& \Rightarrow T & \Rightarrow id_1 \\
& \Rightarrow id_2 & \Rightarrow id_1 \\
& \Rightarrow E + T & \Rightarrow id_1 \\
& \Rightarrow T & \Rightarrow id_1 \\
& \Rightarrow id_2 & \Rightarrow id_2
\end{align*}
\]
A Simple Top-Down Parser

We'll build a rudimentary top-down parser that simply tries each possible expansion of a non-terminal, in order of production definition.

If an expansion leads to a token sequence that doesn’t match the current token being parsed, we backup and try the next possible production choice.

We stop when all the input tokens are correctly matched or when all possible production choices have been tried.

Example

Given the productions

\[ S \rightarrow a \]
\[ \mid ( S ) \]

we try a, then (a), then ((a)), etc.

Let’s next try an additional alternative:

\[ S \rightarrow a \]
\[ \mid ( S ) \]
\[ \mid ( S ] \]

Let’s try to parse a, then [a], then ([a]), etc.

We'll count the number of productions we try for each input.

• For input = a
  We try \( S \rightarrow a \) which works.
  Matches needed = 1

• For input = ( a ]
  We try \( S \rightarrow a \) which fails.
  We next try \( S \rightarrow ( S ) \).
  We expand the inner \( S \) three different ways; all fail.
  Finally, we try \( S \rightarrow ( S ] \).
  The inner \( S \) expands to a, which works.
  Total matches tried = 1 + (1 + 3) + (1 + 1) = 7.

• For input = ( [ a ]
  We try \( S \rightarrow a \) which fails.
  We next try \( S \rightarrow ( S ) \).
  We match the inner \( S \) to (a] using 7 steps, then fail to match the last ].
  Finally, we try \( S \rightarrow ( S ] \).
  We match the inner \( S \) to (a] using 7 steps, then match the last ].
  Total matches tried = 1 + (1 + 17) + (1 + 17) = 37.

• For input = ( [ [ a ]]]
  We try \( S \rightarrow a \) which fails.
  We next try \( S \rightarrow ( S ) \).
  We match the inner \( S \) to ([a]) using 17 steps, then fail to match the last ].
  Finally, we try \( S \rightarrow ( S ] \).
  We match the inner \( S \) to ([a]) using 17 steps, then match the last ].
  Total matches tried = 1 + (1 + 17) + (1 + 17) = 37.

Adding one extra ( ] ] pair doubles the number of matches we need to do the parse.

In fact to parse \([a]\] takes \(5^i2^i-3\) matches. This is exponential growth!
With a more effective *dynamic programming* approach, in which results of intermediate parsing steps are cached, we can reduce the number of matches needed to $n^3$ for an input with $n$ tokens.

Is this acceptable?

No!

Typical source programs have at least 1000 tokens, and $1000^3 = 10^9$ is a lot of steps, even for a fast modern computer.

The solution?

—Smarter selection in the choice of productions we try.

**Reading Assignment**

Read Chapter 5 of *Crafting a Compiler, Second Edition*.

**Prediction**

We want to avoid trying productions that can’t possibly work.

For example, if the current token to be parsed is an identifier, it is useless to try a production that begins with an integer literal.

Before we try a production, we’ll consider the set of terminals it might initially produce. If the current token is in this set, we’ll try the production.

If it isn’t, there is no way the production being considered could be part of the parse, so we’ll ignore it.

A *predict function* tells us the set of tokens that might be initially generated from any production.

---

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

\[
\text{Follow}(A) = \{ a \in V_t \mid S \Rightarrow^+ \ldots A a \ldots \}
\]

In our example,

\[
\text{Follow}(\text{Label} \rightarrow \lambda) = \{ \text{id}, \text{if}, \text{read} \}
\]

(since these terminals can immediately follow uses of Label in the given productions).

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 remove the Label prefix from all the statement productions (now intlit won't predict all four productions).

We now have

\[
\text{Stmt} \rightarrow \text{Label BasicStmt}
\]

\[
\text{BasicStmt} \rightarrow \text{id} = \text{Expr} ;
\]

| if \( \text{Expr} \) then \( \text{Stmt} \) ;
| \( \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

\[
\text{BasicStmt} \rightarrow \text{id StmtSuffix}
\]

\[
\text{StmtSuffix} \rightarrow = \text{Expr} ;
\]

| ( \( \text{Args} ) ;

we no longer have any doubt over which production \( \text{id} \) predicts.

We now have

We now have

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

\[
\text{First}(X_1...X_n) = \{a \in V_t \mid A \rightarrow X_1...X_n \Rightarrow^* a\ldots\}
\]

Follow(A) = \{a \in V_t \mid S \Rightarrow^+ ...Aa\ldots\}

**Example**

<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>

Since the predict sets of both B productions and both D productions are disjoint, this grammar is LL(1).
**Recursive Descent Parsers**

An early implementation of top-down (LL(1)) parsing was recursive descent.

A parser was organized as a set of parsing procedures, one for each non-terminal. Each parsing procedure was responsible for parsing a sequence of tokens derivable from its non-terminal.

For example, a parsing procedure, A, when called, would call the scanner and match a token sequence derivable from A.

Starting with the start symbol’s parsing procedure, we would then match the entire input, which must be derivable from the start symbol.

---

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 → X₁...Xₙ | A → Y₁...Yₘ | ...

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 → X₁...Xₙ | A → Y₁...Yₘ | ...

is

```c
void A() {
    if (currentToken in Predict(A→X₁...Xₙ))
        for(i=1;i<=n;i++)
            if (X[i] is a terminal)
                Match(X[i]);
            else X[i]();
    else
        if (currentToken in Predict(A→Y₁...Yₘ))
            ...
        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><code>Prog → { Stmts } Eof</code></td>
<td>`{</td>
</tr>
<tr>
<td><code>Stmts → Stmt Stmts</code></td>
<td>id if</td>
</tr>
<tr>
<td><code>Stmts → λ</code></td>
<td>}</td>
</tr>
<tr>
<td><code>Stmt → id = Expr ;</code></td>
<td>id</td>
</tr>
<tr>
<td><code>Stmt → if ( Expr ) Stmt</code></td>
<td>if</td>
</tr>
<tr>
<td><code>Expr → id Etail</code></td>
<td>id</td>
</tr>
<tr>
<td><code>Etail → + Expr</code></td>
<td>+</td>
</tr>
<tr>
<td><code>Etail → - Expr</code></td>
<td>-</td>
</tr>
<tr>
<td><code>Etail → λ</code></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(") ");
        Stmt();
    }
}

void Expr() {
    Match(id);
    Etail();
}

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

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

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

<table>
<thead>
<tr>
<th>Calls Pending</th>
<th>Remaining Input</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{Match(&quot;+&quot;)}; \texttt{Expr()}; \texttt{Match(&quot;;&quot;); \texttt{Stmts()}; \texttt{Match(&quot;&quot;)}; \texttt{Match(Eof)}</td>
<td>\texttt{+ c; } Eof</td>
</tr>
<tr>
<td>\texttt{Expr()}; \texttt{Match(&quot;;&quot;); \texttt{Stmts()}; \texttt{Match(&quot;&quot;)}; \texttt{Match(Eof)}</td>
<td>\texttt{c; } Eof</td>
</tr>
<tr>
<td>\texttt{Match(id)}; \texttt{Etail()}; \texttt{Match(&quot;;&quot;); \texttt{Stmts()}; \texttt{Match(&quot;&quot;)}; \texttt{Match(Eof)}</td>
<td>\texttt{c; } Eof</td>
</tr>
<tr>
<td>\texttt{Etail()}; \texttt{Match(&quot;;&quot;); \texttt{Stmts()}; \texttt{Match(&quot;&quot;)}; \texttt{Match(Eof)}</td>
<td>\texttt{;} Eof</td>
</tr>
<tr>
<td>/* null */ \texttt{Match(&quot;;&quot;); \texttt{Stmts()}; \texttt{Match(&quot;&quot;)}; \texttt{Match(Eof)}</td>
<td>\texttt{;} Eof</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Calls Pending</th>
<th>Remaining Input</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{Match(&quot;;&quot;); \texttt{Stmts()}; \texttt{Match(&quot;&quot;)}; \texttt{Match(Eof)}</td>
<td>\texttt{;} Eof</td>
</tr>
<tr>
<td>\texttt{Stmts()}; \texttt{Match(&quot;&quot;)}; \texttt{Match(Eof)}</td>
<td>\texttt{;} Eof</td>
</tr>
<tr>
<td>\texttt{Match(Eof)}</td>
<td>\texttt{Eof}</td>
</tr>
</tbody>
</table>

Done! All input matched
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>
<thead>
<tr>
<th>Calls Pending</th>
<th>Remaining Input</th>
</tr>
</thead>
<tbody>
<tr>
<td>Prog()</td>
<td>{ b + c = a; } Eof</td>
</tr>
<tr>
<td>Match(&quot;{&quot;);</td>
<td></td>
</tr>
<tr>
<td>Stmts();</td>
<td></td>
</tr>
<tr>
<td>Match(&quot;}&quot;);</td>
<td></td>
</tr>
<tr>
<td>Match(Eof);</td>
<td></td>
</tr>
<tr>
<td>Stmts();</td>
<td>b + c = a; } Eof</td>
</tr>
<tr>
<td>Match(&quot;}&quot;);</td>
<td></td>
</tr>
<tr>
<td>Match(Eof);</td>
<td></td>
</tr>
<tr>
<td>Match(id);</td>
<td>b + c = a; } Eof</td>
</tr>
<tr>
<td>Match(=&quot;&quot;);</td>
<td></td>
</tr>
<tr>
<td>Expr();</td>
<td></td>
</tr>
<tr>
<td>Match(&quot;;&quot;);</td>
<td></td>
</tr>
<tr>
<td>Stmts();</td>
<td></td>
</tr>
<tr>
<td>Match(&quot;}&quot;);</td>
<td></td>
</tr>
<tr>
<td>Match(Eof);</td>
<td></td>
</tr>
<tr>
<td>Call to Match fails!</td>
<td>+ c = a; } Eof</td>
</tr>
</tbody>
</table>

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 $\text{Predict}(A \rightarrow X_1...X_n)$
- $T[A][CT] = \text{error}$ 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 $\text{Prog} \rightarrow { \text{Stmts} } \text{Eof}$</td>
<td>${}$</td>
</tr>
<tr>
<td>2 $\text{Stmts} \rightarrow \text{Stmt} \text{Stmts}$</td>
<td>id if</td>
</tr>
<tr>
<td>3 $\text{Stmts} \rightarrow \lambda$</td>
<td>$}$</td>
</tr>
<tr>
<td>4 $\text{Stmt} \rightarrow \text{id} = \text{Expr} ; ; \text{id}$</td>
<td>id</td>
</tr>
<tr>
<td>5 $\text{Stmt} \rightarrow \text{if} ; (\text{Expr}) ; \text{Stmt}$</td>
<td>if</td>
</tr>
<tr>
<td>6 $\text{Expr} \rightarrow \text{id} ; \text{Etail} ; \text{id}$</td>
<td>id</td>
</tr>
<tr>
<td>7 $\text{Etail} \rightarrow + ; \text{Expr}$</td>
<td>+</td>
</tr>
<tr>
<td>8 $\text{Etail} \rightarrow - ; \text{Expr}$</td>
<td>-</td>
</tr>
<tr>
<td>9 $\text{Etail} \rightarrow \lambda$</td>
<td>$}$</td>
</tr>
</tbody>
</table>

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

### LL(1) Parser Driver

Here is the driver we'll use with the LL(1) parse table. We'll also use a *parse stack* that remembers symbols we have yet to match.

```c
void LLDriver(){
  Push(StartSymbol);
  while(! stackEmpty()){ //Let X=Top symbol on parse stack
    //Let CT = current token to match
    if (isTerminal(X)) {
      match(X); //CT is updated
      pop(); //X is updated
    } else if (T[X][CT] != Error){
      //Let T[X][CT] = X\rightarrow Y_1...Y_m
      Replace X with
      Y_1...Y_m on parse stack
    } else SyntaxError(CT);
  }
}
```
Example of LL(1) Parsing

We'll again parse

{ a = b + c; } Eof

We start by placing Prog (the start

symbol) on the parse stack.

<table>
<thead>
<tr>
<th>Parse Stack</th>
<th>Remaining Input</th>
</tr>
</thead>
<tbody>
<tr>
<td>Prog</td>
<td>{ a = b + c; } Eof</td>
</tr>
<tr>
<td>{</td>
<td>{ a = b + c; } Eof</td>
</tr>
<tr>
<td>Stmts</td>
<td>a = b + c; } Eof</td>
</tr>
<tr>
<td>}</td>
<td>a = b + c; } Eof</td>
</tr>
<tr>
<td>Eof</td>
<td>a = b + c; } Eof</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Parse Stack</th>
<th>Remaining Input</th>
</tr>
</thead>
<tbody>
<tr>
<td>id</td>
<td>a = b + c; } Eof</td>
</tr>
<tr>
<td>=</td>
<td>a = b + c; } Eof</td>
</tr>
<tr>
<td>Expr</td>
<td>= b + c; } Eof</td>
</tr>
<tr>
<td>;</td>
<td>= b + c; } Eof</td>
</tr>
<tr>
<td>Stmts</td>
<td>b + c; } Eof</td>
</tr>
<tr>
<td>}</td>
<td>b + c; } Eof</td>
</tr>
<tr>
<td>Eof</td>
<td>b + c; } Eof</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Parse Stack</th>
<th>Remaining Input</th>
</tr>
</thead>
<tbody>
<tr>
<td>Etail</td>
<td>+ c; } Eof</td>
</tr>
<tr>
<td>;</td>
<td>+ c; } Eof</td>
</tr>
<tr>
<td>Stmts</td>
<td>+ c; } Eof</td>
</tr>
<tr>
<td>}</td>
<td>+ c; } Eof</td>
</tr>
<tr>
<td>Eof</td>
<td>+ c; } Eof</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Parse Stack</th>
<th>Remaining Input</th>
</tr>
</thead>
<tbody>
<tr>
<td>Expr</td>
<td>c; } Eof</td>
</tr>
<tr>
<td>;</td>
<td>c; } Eof</td>
</tr>
<tr>
<td>Stmts</td>
<td>c; } Eof</td>
</tr>
<tr>
<td>}</td>
<td>c; } Eof</td>
</tr>
<tr>
<td>Eof</td>
<td>c; } Eof</td>
</tr>
<tr>
<td>id</td>
<td>c; } Eof</td>
</tr>
<tr>
<td>Etail</td>
<td>c; } Eof</td>
</tr>
<tr>
<td>;</td>
<td>c; } Eof</td>
</tr>
<tr>
<td>Stmts</td>
<td>c; } Eof</td>
</tr>
<tr>
<td>}</td>
<td>c; } Eof</td>
</tr>
<tr>
<td>Eof</td>
<td>c; } Eof</td>
</tr>
<tr>
<td>Eof</td>
<td>c; } Eof</td>
</tr>
<tr>
<td>Eof</td>
<td>c; } Eof</td>
</tr>
<tr>
<td>Done!</td>
<td>c; } Eof</td>
</tr>
<tr>
<td>All input matched</td>
<td>c; } Eof</td>
</tr>
</tbody>
</table>
Syntax Errors in LL(1) Parsing

In LL(1) parsing, syntax errors are automatically detected as soon as the first illegal token is seen.

How? When an illegal token is seen by the parser, either it fetches an error entry from the LL(1) parse table or it fails to match an expected token.

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

```
{ b + c = a; } Eof
```

(Where should the first syntax error be detected?)

### Parse Stack

<table>
<thead>
<tr>
<th>Parse Stack</th>
<th>Remaining Input</th>
</tr>
</thead>
<tbody>
<tr>
<td>{ b + c = a; } Eof</td>
<td>{ b + c = a; } Eof</td>
</tr>
<tr>
<td>{ Stmts } Eof</td>
<td>b + c = a; } Eof</td>
</tr>
<tr>
<td>Stmt Stmts } Eof</td>
<td>b + c = a; } Eof</td>
</tr>
<tr>
<td>id = Expr ; Stmts } Eof</td>
<td>b + c = a; } Eof</td>
</tr>
</tbody>
</table>

Current token (+) fails to match expected token (=)!

### Parse Stack

<table>
<thead>
<tr>
<th>Parse Stack</th>
<th>Remaining Input</th>
</tr>
</thead>
<tbody>
<tr>
<td>= Expr ; Stmts } Eof</td>
<td>+ c = a; } Eof</td>
</tr>
</tbody>
</table>

How do LL(1) Parsers Build Syntax Trees?

So far our LL(1) parser has acted like a recognizer. It verifies that input tokens are syntactically correct, but it produces no output.

Building complete (concrete) parse trees automatically is fairly easy.

As tokens and non-terminals are matched, they are pushed onto a second stack, the **semantic stack**.

At the end of each production, an action routine pops off n items from the semantic stack (where n is the length of the production’s right-hand side). It then builds a syntax tree whose root is the
left hand side, and whose children are the $n$ items just popped off.

For example, for production
\[
\text{Stmt} \rightarrow \text{id} = \text{Expr} \; ;
\]
the parser would include an action symbol after the ";" whose actions are:
\[
\begin{align*}
P_4 &= \text{pop}(); \quad \text{// Semicolon token} \\
P_3 &= \text{pop}(); \quad \text{// Syntax tree for Expr} \\
P_2 &= \text{pop}(); \quad \text{// Assignment token} \\
P_1 &= \text{pop}(); \quad \text{// Identifier token} \\
\text{Push}(\text{new StmtNode}(P_1,P_2,P_3,P_4));
\end{align*}
\]

Creating Abstract Syntax Trees

Recall that we prefer that parsers generate abstract syntax trees, since they are simpler and more concise.

Since a parser generator can’t know what tree structure we want to keep, we must allow the user to define “custom” action code, just as Java CUP does.

We allow users to include “code snippets” in Java or C. We also allow labels on symbols so that we can refer to the tokens and trees we wish to access. Our production and action code will now look like this:
\[
\text{Stmt} \rightarrow \text{id} : \text{i} = \text{Expr} : \text{e} ;
\]
\[
\{ : \text{RESULT} = \text{new StmtNode}(i,e); : \}
\]

How do We Make Grammars LL(1)?

Not all grammars are LL(1); sometimes we need to modify a grammar’s productions to create the disjoint Predict sets LL(1) requires.

There are two common problems in grammars that make unique prediction difficult or impossible:

   Two or more productions with the same left hand side begin with the same symbol(s).
   For example,
   \[
   \begin{align*}
   \text{Stmt} &\rightarrow \text{id} = \text{Expr} \; ; \\
   \text{Stmt} &\rightarrow \text{id} (\text{Args}) ;
   \end{align*}
   \]

2. Left-Recursion
   A production of the form
   \[
   \text{A} \rightarrow \text{A} ...
   \]
   is said to be left-recursive.
   When a left-recursive production is used, a non-terminal is immediately replaced by itself (with additional symbols following).
   Any grammar with a left-recursive production can never be LL(1).
   Why?
   Assume a non-terminal $A$ reaches the top of the parse stack, with $CT$ as the current token. The LL(1) parse table entry, $T[A][CT]$, predicts $A \rightarrow A ...$
   We expand $A$ again, and $T[A][CT]$, so we predict $A \rightarrow A ...$ again. We are in an infinite prediction loop!