**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

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

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

\[ \text{Predict}(A \rightarrow X_1...X_n) = \]

If \( X_1...X_n \Rightarrow^* \lambda \)

Then First(X_1...X_n) U Follow(A)

Else First(X_1...X_n)

If some CFG, G, has the property that for all pairs of distinct productions with the same lefthand side, \( A \rightarrow X_1...X_n \) and \( A \rightarrow Y_1...Y_m \) it is the case that

\[ \text{Predict}(A \rightarrow X_1...X_n) \cap \text{Predict}(A \rightarrow Y_1...Y_m) = \phi \]

then G is LL(1).

LL(1) grammars are easy to parse in a top-down manner since predictions are always correct.

**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:

```java
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:

\[
A \rightarrow X_1 \ldots X_n \mid A \rightarrow Y_1 \ldots Y_m \mid ...
\]

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

\[
A \rightarrow X_1 \ldots X_n \mid A \rightarrow Y_1 \ldots Y_m \mid ...
\]

is

```java
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>$Prog \rightarrow { \text{Stmts} } \text{Eof}$</td>
<td>${ }$</td>
</tr>
<tr>
<td>$\text{Stmts} \rightarrow \text{Stmt} \text{Stmts}$</td>
<td>$\text{id if}$</td>
</tr>
<tr>
<td>$\text{Stmts} \rightarrow \lambda$</td>
<td>${ }$</td>
</tr>
<tr>
<td>$\text{Stmt} \rightarrow \text{id = Expr ;}$</td>
<td>$\text{id}$</td>
</tr>
<tr>
<td>$\text{Stmt} \rightarrow \text{if ( Expr ) Stmt}$</td>
<td>$\text{if}$</td>
</tr>
<tr>
<td>$\text{Expr} \rightarrow \text{id Etail}$</td>
<td>$\text{id}$</td>
</tr>
<tr>
<td>$\text{Etail} \rightarrow + \text{ Expr}$</td>
<td>$+$</td>
</tr>
<tr>
<td>$\text{Etail} \rightarrow - \text{ Expr}$</td>
<td>$-$</td>
</tr>
<tr>
<td>$\text{Etail} \rightarrow \lambda$</td>
<td>$\lambda$</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(")");
        Match(Eof);
    }
}
```

Let's use recursive descent to parse

\[
\{ \ a = b + c; \ } \text{ Eof}
\]

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>$Prog()$</td>
<td>$( a = b + c; ) \text{ Eof}$</td>
</tr>
<tr>
<td>Match(&quot;(&quot;);</td>
<td>$( a = b + c; ) \text{ Eof}$</td>
</tr>
<tr>
<td>Stmts();</td>
<td>$( a = b + c; ) \text{ Eof}$</td>
</tr>
<tr>
<td>Match(&quot;*&quot;);</td>
<td>$a = b + c; \text{ Eof}$</td>
</tr>
<tr>
<td>Match(Eof);</td>
<td>$a = b + c; \text{ Eof}$</td>
</tr>
<tr>
<td>Stmts();</td>
<td>$a = b + c; \text{ Eof}$</td>
</tr>
<tr>
<td>Match(&quot;(&quot;);</td>
<td>$( a = b + c; ) \text{ Eof}$</td>
</tr>
<tr>
<td>Expr();</td>
<td>$( a = b + c; ) \text{ Eof}$</td>
</tr>
<tr>
<td>Match(&quot;;&quot;);</td>
<td>$( a = b + c; ) \text{ Eof}$</td>
</tr>
<tr>
<td>Match(Eof);</td>
<td>$( a = b + c; ) \text{ Eof}$</td>
</tr>
<tr>
<td>Match(id);</td>
<td>$( a = b + c; ) \text{ Eof}$</td>
</tr>
<tr>
<td>Match(&quot;=&quot;);</td>
<td>$( a = b + c; ) \text{ Eof}$</td>
</tr>
<tr>
<td>Expr();</td>
<td>$( a = b + c; ) \text{ Eof}$</td>
</tr>
<tr>
<td>Match(&quot;*&quot;);</td>
<td>$( a = b + c; ) \text{ Eof}$</td>
</tr>
<tr>
<td>Stmts();</td>
<td>$( a = b + c; ) \text{ Eof}$</td>
</tr>
<tr>
<td>Match(&quot;)&quot;);</td>
<td>$( a = b + c; ) \text{ Eof}$</td>
</tr>
<tr>
<td>Match(Eof);</td>
<td>$( a = b + c; ) \text{ Eof}$</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 $\text{Predict}(A \rightarrow X_1...X_n)$

$$T[A][CT] = \text{error}$$  
if $CT$ predicts no production with $A$ as its lefthand side

**CSX-lite Example**

### Production Predict Set

- **Prog** $\rightarrow$ (Stmts) Eof
- **Stmts** $\rightarrow$ Stmt Stmts id if
- **Stmts** $\rightarrow$ \lambda
- **Stmt** $\rightarrow$ if (Expr) Stmt if
- **Expr** $\rightarrow$ id Etail id
- **Etail** $\rightarrow$ + Expr +
- **Etail** $\rightarrow$ - Expr -
- **Etail** $\rightarrow$ ( ) ;

<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 id if</td>
<td>2</td>
</tr>
<tr>
<td>3 Stmts $\rightarrow$ \lambda</td>
<td>4</td>
</tr>
<tr>
<td>4 Stmt $\rightarrow$ id = Expr ; id</td>
<td>7</td>
</tr>
<tr>
<td>5 Stmt $\rightarrow$ if (Expr) Stmt if</td>
<td>8</td>
</tr>
<tr>
<td>6 Expr $\rightarrow$ id Etail id</td>
<td>9</td>
</tr>
<tr>
<td>7 Etail $\rightarrow$ + Expr +</td>
<td>7</td>
</tr>
<tr>
<td>8 Etail $\rightarrow$ - Expr -</td>
<td>8</td>
</tr>
<tr>
<td>9 Etail $\rightarrow$ ( ) ;</td>
<td>9</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>)</td>
</tr>
<tr>
<td>Stmts</td>
<td>3</td>
</tr>
<tr>
<td>Stmt</td>
<td>5</td>
</tr>
<tr>
<td>Expr</td>
<td>6</td>
</tr>
<tr>
<td>Etail</td>
<td>9</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 | Remaining Input
--- | ---
Prog | { b + c = a; } Eof
(Stmts)
Eof | { b + c = a; } Eof
Stmts
Eof | b + c = a; } Eof
Stmt
Stmts
Eof | b + c = a; } Eof
id
Expr
; 
Stmts
Eof | b + c = a; } Eof

### Parse Stack | Remaining Input
--- | ---
= 
Expr | + c = a; } Eof
; 
Stmts
} 
Eof | + c = a; } Eof

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

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