**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:

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:

\[
\begin{align*}
T & \Rightarrow E \Rightarrow \\
& \downarrow \\
& id_1 \\
E & \Rightarrow T \Rightarrow \\
& \downarrow \\
& id_1 \\
T & \Rightarrow E \Rightarrow \\
& \downarrow \\
& id_1 \\
& \downarrow \\
& id_2
\end{align*}
\]

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+8)+(1+8) = 17.
• 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 $[^{i}a^{i}]$ takes $5*2^{i} - 3$ 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
Get and read Chapter 5 of Crafting a Compiler featuring Java.
(Available from Dolt Tech Store)

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 will tell us the set of tokens that might be initially generated from any production.
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 \text{ in } V_t \mid A \rightarrow X_1...X_n \Rightarrow^* a...\} 

For example, given 
Stmt \rightarrow \text{Label id = Expr ;} 
\mid \text{Label if Expr then Stmt ;} 
\mid \text{Label read ( IdList ) ;} 
\mid \text{Label id ( Args ) ;} 
Label \rightarrow \text{intlit :} 
\mid \lambda 

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 \text{ in } V_t \mid S \Rightarrow^+ ...Aa...\}

In our example,
\text{Follow(Label \rightarrow \lambda)} = \{ \text{id, if, read } \}

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
Stmt \rightarrow Label BasicStmt 
BasicStmt \rightarrow id = Expr ; 
\mid \text{if Expr then Stmt ;} 
\mid \text{read ( IdList ) ;} 
\mid \text{id ( Args ) ;} 
Label \rightarrow \text{intlit :} 
\mid \lambda 

Now id predicts two different BasicStmt productions. If we rewrite these two productions into
BasicStmt \rightarrow id StmtSuffix 
StmtSuffix \rightarrow = Expr ; 
\mid ( \text{Args} ) ; 

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

Now let's parse
id ( intlit ) ;

Our start symbol is Stmt and the initial token is id.

id can predictStmt \rightarrow Label id = Expr ;
id then predictsLabel \rightarrow \lambda

The id is matched, but "(" doesn't match "=" so be backup and try a different production for Stmt.

id also predictsStmt \rightarrow Label id ( Args ) ;
Again, 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.
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⇒* λ
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...Yₗ
it is the case that
Predict(A → X₁...Xₙ) ∩
Predict(A → Y₁...Yₗ) = Ø
then G is LL(1).

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

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>

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).