How does JavaCup Work?

The main limitation of LL(1) parsing is that it must predict the correct production to use when it first starts to match the production's right-hand side. An improvement to this approach is the LALR(1) parsing method that is used in JavaCUP (and Yacc and Bison too). The LALR(1) parser is bottom-up in approach. It tracks the portion of a right-hand side already matched as tokens are scanned. It may not know immediately which is the correct production to choose, so it tracks sets of possible matching productions.

Since we may not know which production will eventually be fully matched, we may need to track a configuration set. A configuration set is sometimes called a state. When we predict a production, we place the “dot” at the beginning of a production:

\[ X \rightarrow \lambda \]

This indicates that the production may possibly be matched, but no symbols have actually yet been matched.

We may predict a \( \lambda \)-production:

\[ X \rightarrow \lambda \]

When a \( \lambda \)-production is predicted, it is immediately matched, since \( \lambda \) can be matched at any time.

Configurations

We'll use the notation

\[ X \rightarrow A B \cdot C D \]

to represent the fact that we are trying to match the production

\[ X \rightarrow A B \cdot C D \] with A and B matched so far.

A production with a “\( \cdot \)” somewhere in its right-hand side is called a configuration.

Our goal is to reach a configuration with the “dot” at the extreme right:

\[ X \rightarrow A B C D \cdot \]

This indicates that an entire production has just been matched.
**Starting the Parse**

At the start of the parse, we know some production with the start symbol must be used initially. We don’t yet know which one, so we predict them *all*:

\[
\begin{align*}
S & \rightarrow \quad \bullet \ A \ B \ C \ D \\
S & \rightarrow \quad \bullet \ e \ F \ g \\
S & \rightarrow \quad \bullet \ h \ I \\
& \ldots
\end{align*}
\]

**Closure**

When we encounter a configuration with the dot to the left of a non-terminal, we know we need to try to match that non-terminal. Thus in

\[
X \rightarrow \quad \bullet \ A \ B \ C \ D
\]

we need to match some production with A as its left hand side.

Which production? We don’t know, so we predict *all* possibilities:

\[
\begin{align*}
A & \rightarrow \quad \bullet \ P \ Q \ R \\
A & \rightarrow \quad \bullet \ s \ T \\
& \ldots
\end{align*}
\]

The newly added configurations may predict other non-terminals, forcing additional productions to be included. We continue this process until no additional configurations can be added. This process is called closure (of the configuration set).

Here is the closure algorithm:

```c
ConfigSet Closure(ConfigSet C)
{
    repeat
        if \((X \rightarrow a \ B \ d \ is \ in \ C \ &\&
             B \ is \ a \ non-terminal)\)
            Add all configurations of
            the form \(B \rightarrow \bullet \ g \ to \ C)\)
        until (no more configurations
            can be added);
    return C;
}
```

**Example of Closure**

Assume we have the following grammar:

\[
\begin{align*}
S & \rightarrow \quad A \ b \\
A & \rightarrow \quad C \ D \\
C & \rightarrow \quad D \\
C & \rightarrow \quad c \\
D & \rightarrow \quad d
\end{align*}
\]

To compute Closure(S \(\rightarrow \bullet \ A \ b\)) we first include all productions that rewrite A:

\[
\begin{align*}
A & \rightarrow \quad \bullet \ C \ D \\
C & \rightarrow \quad \bullet \ d
\end{align*}
\]

Now C productions are included:

\[
\begin{align*}
C & \rightarrow \quad \bullet \ D \\
C & \rightarrow \quad \bullet \ c
\end{align*}
\]
Finally, the D production is added:
\[ D \rightarrow \ast d \]
The complete configuration set is:
\[
\begin{align*}
S & \rightarrow \ast A b \\
A & \rightarrow \ast C D \\
C & \rightarrow \ast D \\
C & \rightarrow \ast c \\
D & \rightarrow \ast d
\end{align*}
\]
This set tells us that if we want to match an \( A \), we will need to match a \( C \), and this is done by matching a \( c \) or \( d \) token.

**Shift Operations**

When we match a symbol (a terminal or non-terminal), we *shift* the “dot” past the symbol just matched. Configurations that don’t have a dot to the left of the matched symbol are deleted (since they didn’t correctly anticipate the matched symbol).

The \texttt{GoTo} function computes an updated configuration set after a symbol is shifted:

\[
\text{ConfigSet GoTo(ConfigSet C,Symbol X)}\{
\varnothing
\text{for each configuration } f \text{ in } C \\
\text{if } (f \text{ is of the form } A \rightarrow \alpha X \beta) \\
\text{Add } A \rightarrow \alpha X \ast \beta \text{ to } B; \\
\text{return Closure}(B); \\
\}
\]

For example, if \( C \) is
\[
\begin{align*}
S & \rightarrow \ast A b \\
A & \rightarrow \ast C D \\
C & \rightarrow \ast D \\
C & \rightarrow \ast c \\
D & \rightarrow \ast d
\end{align*}
\]
and \( x \) is \( C \), then \texttt{GoTo} returns
\[
\begin{align*}
A & \rightarrow \ast C D \\
D & \rightarrow \ast d
\end{align*}
\]

**Reduce Actions**

When the dot in a configuration reaches the rightmost position, we have matched an entire right-hand side. We are ready to replace the right-hand side symbols with the left-hand side of the production. The left-hand side symbol can now be considered matched.

If a configuration set can shift a token and also reduce a production, we have a potential *shift/reduce error*.

If we can reduce more than one production, we have a potential *reduce/reduce error*.

How do we decide whether to do a shift or reduce? How do we choose among more than one reduction?
We examine the next token to see if it is consistent with the potential reduce actions. The simplest way to do this is to use Follow sets, as we did in LL(1) parsing.

If we have a configuration

\[ A \rightarrow \alpha \cdot \]

we will reduce this production only if the current token, \( CT \), is in \( \text{Follow}(A) \).

This makes sense since if we reduce \( \alpha \) to \( A \), we can't correctly match \( CT \) if \( CT \) can't follow \( A \).

---

### Shift/Reduce and Reduce/Reduce Errors

If we have a parse state that contains the configurations

\[ A \rightarrow \alpha \cdot \]
\[ B \rightarrow \beta \cdot a \gamma \]

and \( a \) in \( \text{Follow}(A) \) then there is an unresolvable shift/reduce conflict. This grammar can't be parsed.

Similarly, if we have a parse state that contains the configurations

\[ A \rightarrow \alpha \cdot \]
\[ B \rightarrow \beta \cdot \]

and \( \text{Follow}(A) \cap \text{Follow}(B) \neq \emptyset \), then the parser has an unresolvable reduce/reduce conflict. This grammar can't be parsed.

---

### Building Parse States

All the manipulations needed to build and complete configuration sets suggest that parsing may be slow—configuration sets need to be updated after each token is matched.

Fortunately, all the configuration sets we ever will need can be computed and tabled in advance, when a tool like Java Cup builds a parser.

The idea is simple. We first compute an initial parse state, \( s_0 \), that corresponds to predicting productions that expand the start symbol. We then just compute successor states for each token that might be scanned. A complete set of states can be computed. For typical programming language grammars, only a few hundred states are needed.

Here is the algorithm that builds a complete set of parse states for a grammar:

```java
StateSet BuildStates()
{
    Let \( s_0 = \text{Closure}(\{S \rightarrow \cdot \alpha, S \rightarrow \cdot \beta, \ldots\}) \); \n    C={s_0};
    while (not all states in C are marked){
        Choose any unmarked state, s, in C
        Mark s;
        For each X in terminals U nonterminals {
            if (GoTo(s,X) is not in C)
                Add GoTo(s,X) to C;
        }
    }
    return C;
}
```
**Configuration Sets for CSX-Lite**

<table>
<thead>
<tr>
<th>State</th>
<th>Configuration Set</th>
</tr>
</thead>
<tbody>
<tr>
<td>$s_0$</td>
<td>Prog → • (Stmts) Eof</td>
</tr>
<tr>
<td>$s_1$</td>
<td>Prog → • (Stmts) Eof</td>
</tr>
<tr>
<td>$s_2$</td>
<td>Prog → • (Stmts • Eof)</td>
</tr>
<tr>
<td>$s_3$</td>
<td>Prog → • (Stmts • Eof)</td>
</tr>
<tr>
<td>$s_4$</td>
<td>Stmt → id * = Expr ;</td>
</tr>
<tr>
<td>$s_5$</td>
<td>Stmt → if * (Expr) Stmt</td>
</tr>
</tbody>
</table>

**State Configuration Set**

<table>
<thead>
<tr>
<th>State</th>
<th>Configuration Set</th>
</tr>
</thead>
<tbody>
<tr>
<td>$s_6$</td>
<td>Prog → {Stmts} • Eof</td>
</tr>
<tr>
<td>$s_7$</td>
<td>Stmts → Stmt Stmts •</td>
</tr>
<tr>
<td>$s_8$</td>
<td>Stmts → Stmt Stmts</td>
</tr>
<tr>
<td>$s_9$</td>
<td>Stmts → Stmt Stmts</td>
</tr>
<tr>
<td>$s_{10}$</td>
<td>Stmts → Stmt Stmts</td>
</tr>
<tr>
<td>$s_{11}$</td>
<td>Stmts → Stmt Stmts</td>
</tr>
<tr>
<td>$s_{12}$</td>
<td>Stmts → Stmt Stmts</td>
</tr>
<tr>
<td>$s_{13}$</td>
<td>Stmts → Stmt Stmts</td>
</tr>
</tbody>
</table>

**Parser Action Table**

We will table possible parser actions based on the current state (configuration set) and token. Given configuration set C and input token T four actions are possible:

- **Reduce i**: The i-th production has been matched.
- **Shift**: Match the current token.
- **Accept**: Parse is correct and complete.
- **Error**: A syntax error has been discovered.
We will let \( A[C][T] \) represent the possible parser actions given configuration set \( C \) and input token \( T \).

\[
A[C][T] = \{ \text{Reduce } i \mid \text{ } i\text{-th production is } A \rightarrow \alpha \\
\text{and } A \rightarrow \alpha \ast \text{ is in } C \\
\text{and } T \text{ in } \text{Follow}(A) \}\ 
U \text{ (If } (B \rightarrow \beta \ast T \gamma \text{ is in } C) \}
\{\text{Shift} \text{ else } \phi\}
\]

This rule simply collects all the actions that a parser might do given \( C \) and \( T \).

But we want parser actions to be unique so we require that the parser action always be \textit{unique} for any \( C \) and \( T \).

If the parser action isn't unique, then we have a shift/reduce error or reduce/reduce error. The grammar is then rejected as unparsable.

If parser actions are always unique then we will consider a shift of EOF to be an accept action.

An empty (or undefined) action for \( C \) and \( T \) will signify that token \( T \) is illegal given configuration set \( C \).

A syntax error will be signaled.