Shift-Reduce parsing

Intuitive view:

- *Shift* the input cursor past matching symbols, keeping a marker in each partially matched production
- *Reduce* matching symbols to non-terminal when end of production is reached; slide the input cursor back to match non-terminals and terminals

Implementation:

- Build state machine for *all* possible paths through productions
• Interpret as push-down automaton: \textit{reduce} action pops earlier position
A trivial grammar

Comma-separated lists (just one level)

\[
\langle S \rangle ::= \langle L \rangle \ \$ \\
\langle L \rangle ::= \langle L \rangle \ , \ \ i \\
\langle L \rangle ::= \ i \\
\]

Note left recursion — not a problem

Examples:

a, b, c \$ \\
a \$
LR(0) parsing table

Don’t worry about how we built it (yet)
Initial Configuration
Step 1: Shift 1 (i)

\[
S ::= \cdot L \$
\]

\[
L ::= \cdot L, i
\]

\[
L ::= \cdot i
\]

\[
S ::= L \$
\]

\[
L ::= i
\]

\[
L ::= L, i
\]

\[
S ::= L \$
\]

\[
L ::= i
\]

\[
L ::= L, i
\]

\[
S ::= L \$
\]

\[
L ::= i
\]

\[
L ::= L, i
\]

\[
S ::= L \$
\]

\[
L ::= i
\]

\[
L ::= L, i
\]
Step 2: Reduce \( L \)
Step 3: Shift 1 (L)
Step 4: Shift 2 (,)

U. Oregon CIS 461/561, Winter 2005
Step 5: Shift 5 (i)
Step 6: Reduce $L$

```
S ::= • L $
L ::= • L, i
L ::= • i
```

U. Oregon CIS 461/561, Winter 2005
Step 7: Shift 2 (,)
S ::= L $
L ::= L , i
L ::= i
S ::= L $L ::= L , i
S ::= L $
L ::= i
L ::= L , i
L ::= L , i
S ::= L $S ::= L $L ::= L , i
L ::= L , i
L ::= L , i
S ::= L $
Step 8: Shift 2 (,)
Step 9: Shift 5 (i)
\[ S ::= L \$
\]
\[ L ::= L , i \\
L ::= i 
\]

The diagram illustrates the production rules for the grammar:

1. \[ S ::= L $ \]
2. \[ S ::= L \$ \]
3. \[ S ::= L , i \]
4. \[ L ::= i \]
5. \[ L ::= L , i \]
6. \[ L ::= L , i \$
\]

The diagram shows the transition from one state to another, with arrows indicating the production rules.
Step 10: Reduce L
Step 11: Shift 1 (L)
S ::= L $
L ::= L , i 
L ::= i 
S ::= L $
L ::= i 
L ::= L , i

U. Oregon CIS 461/561, Winter 2005
Step 12: Shift 2 ($)
Step 13: Reduce S
S ::= L $
L ::= L , i
L ::= i
Shift-Reduce Parsing Tables

Core representation used by all LR parsing techniques:

- Position in production — called LR(0) “item”
- State = set of items (multiple partial matches)
- Pre-built state machine for all possible paths

Difference: Use of lookahead to allow reduction

- LR(0): No lookahead (too weak)
• SLR(1): Use “first” and “follow” as in LL(1)

• LR(1): Compute lookaheads\textsuperscript{1} and split states (too expensive)

• LALR(1): Compute lookaheads, but don’t split states

\textsuperscript{1}Method described later
Building SLR parsing tables

For SLR(1) and LALR(1), we start with LR(0) “items”

Treat productions as NFA, LR(0) items as NFA states, sets of items as DFA states.

(The subset construction, again)
Comma list grammar as NFA

\[
\langle S \rangle ::= \langle L \rangle \; \$
\]

\[
\langle L \rangle ::= \langle L \rangle \; , \; i
\]

\[
\langle L \rangle ::= \langle i \rangle
\]
Grammar NFA notes

• $\epsilon$ transitions for each non-terminal

• “Final” state means: reduce
LR(0) machine (CFSM)

Similar to subset construction (NFA to DFA), except

• State is a set of *items*
  Item is a production with a marker

• Transitions on each possible next symbol
  *shift* and *goto*

• *Reduce* action when marker reaches end of production
LR(0) CFSM, Comma lists

\[ S ::= L \$
\]
\[ L ::= L\ , \ i
\]
\[ L ::= i
\]

(Build cfsm on chalkboard)
LR(0) CFSM, Comma lists

\[ \langle S \rangle ::= \langle L \rangle \, \$ \]

\[ \langle L \rangle ::= \langle L \rangle \, , \, i \]

\[ \langle L \rangle ::= \langle i \rangle \]
What about the Goto function?

Goto is just a short-cut —

Instead of shifting non-terminal after reduction, use goto function immediately.

CFSM terminal edges are “shift”, non-terminal edges are “goto”

Example: Goto[1,L] is 2
Ambiguity in shift-reduce parsing

\[ S ::= E \ $ \]

Consider:

\[ E ::= E + E \]
\[ E ::= i \]

*(Build cfsm on chalkboard)*
Shift-reduce conflict
Conflict (shift-reduce): What do we do on +?

It’s a real ambiguity: two different derivations of \( i + i + i \)
Removing ambiguity

\[ S ::= E \]$ 
\[ E ::= E + i \] 
\[ E ::= i \]

*Note: left recursion is not a problem for LR parsing*

*(Build cfsm on chalkboard)*
Choosing productions

LL(1) parsers must make choice with the first token in a RHS.
LR(0) parsers look at the whole RHS, then decide. xLR(1) look one token beyond.

- SLR(1): Compute lookahead as follow set from the grammar
- LALR(1): Compute refined follow set from a path through the grammar (path to current state).
- LR(1): Compute refined follow set from a path, and distinguish states by their follow set.

Question: Why is lookahead used only for reduce actions?
Adding SLR(1) lookahead

Comma list grammar is LR(0) (no shift/reduce conflicts)

In general, lookahead is necessary, even for non-ambiguous grammars.

SLR solution: Reduce if next token in “follow” set of non-terminal (or entire rhs).
SLR(1) lookahead for comma lists

Follow(L) is comma or $
Follow(S) is $
Capturing precedence in SLR(1) lookahead

\[ \langle S \rangle ::= \langle E \rangle \ \$ \]
\[ \langle E \rangle ::= \langle E \rangle + \langle T \rangle \]
\[ \langle E \rangle ::= \ T \]
\[ \langle T \rangle ::= \langle T \rangle \ast \ i \]
\[ \langle T \rangle ::= \ i \]

(Build cfsm on chalkboard, tracing \( E+T*i \)
Use “follow” set to resolve shift/reduce conflict)
LR(1) lookahead

LR(0) items construction \textit{plus}

\begin{itemize}
  \item Lookahead of start symbol is $\$
  \item Lookahead propagates \textit{except} for closure
  \item In closure, we pick up the \textit{following} symbol
\end{itemize}

Note that LR(1) lookahead for $A$ will always be a subset of \textit{Follow}(A)
A (simplified) problem from C

\[ \langle \text{decl} \rangle ::= \langle \text{func-decl} \rangle \mid \langle \text{var-decl} \rangle \]

\[ \langle \text{var-decl} \rangle ::= \langle \text{var-type} \rangle \langle \text{var-list} \rangle ; \]

\[ \langle \text{var-type} \rangle ::= \text{id} \]

\[ \langle \text{var-list} \rangle ::= \langle \text{var-list} \rangle , \text{id} \mid \text{id} \]

\[ \langle \text{func-decl} \rangle ::= \langle \text{func-type} \rangle \langle \text{func-name} \rangle \]

\[ ( \langle \text{args} \rangle ) ; \]

\[ \langle \text{func-type} \rangle ::= \text{id} \mid \text{void} \]

\[ \langle \text{func-name} \rangle ::= \text{id} \]

(Build cfsm on chalkboard; requires more than one token of lookahead. Fix
grammar to use $\text{; }$, $( )$, and $\text{, }$ to resolve reduce/reduce conflict.
A historical note . . .

Until early 70’s, precedence parsing was common.

LR(0) was too weak, LR(1) was too expensive

In early 70’s, most switched to SLR(1) and LALR(1) (but a few stayed with LL(1) for performance and error recovery)

LR(k) may eventually become practical, as LL(k) has, but don’t hold your breath.