Shift-Reduce (bottom-up) Parsing

Michal Young
University of Oregon
CIS 461/561

Spring 2001
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: *reduce* action pops earlier position
A trivial grammar

Comma-separated lists (just one level)

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

Note left recursion — not a problem

Examples:

a, b, c $

a $

CIS 461/561, Spring 2001
LR(0) parsing table

Don’t worry about how we built it (yet)
Initial Configuration

1. $S ::= \cdot L \cdot$
   - $L ::= \cdot L, i$
   - $L ::= \cdot i$

2. $S ::= L \cdot$
   - $L ::= L, i$
   - $L ::= \cdot i$

3. $S ::= L \cdot$

4. $L ::= \cdot i$

5. $L ::= L, \cdot i$

6. $L ::= L, \cdot i$

- $a, b, c$
Step 1: Shift 1 (i)

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

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

\[
L ::= \cdot i
\]

\[
S ::= L$
\]

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

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

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

$a, b, c \$

CIS 461/561, Spring 2001
Step 2: Reduce $L$

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

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

\[
L ::= \cdot i
\]
Step 3: Shift 1 (L)

1. $S ::= \cdot L$
   - $L ::= \cdot, i$
   - $L ::= \cdot i$

2. $S ::= L \cdot$
   - $L ::= L \cdot, i$

3. $S ::= L$

4. $L ::= i \cdot$

5. $L ::= L \cdot i$

6. $L ::= L, i \cdot$

The diagram shows a transition from $S ::= \cdot L$ to $S ::= L$, then $L ::= \cdot$ to $L ::= L \cdot$, and finally $L ::= i \cdot$ to $L ::= L, i \cdot$. The symbols $\cdot, b, c$ and $a$ are also shown at different stages of the transition.
Step 4: Shift 2 (,)

1. S ::= • L $
   L ::= • L, i
   L ::= • i$

2. S ::= L • $
   S ::= • L, i

3. S ::= L $

4. L ::= • i

5. L ::= L •

6. L ::= L, • i

CIS 461/561, Spring 2001
Step 5: Shift 5 (i)

L ::= L, i
L ::= L, i
L ::= i
S ::= L
S ::= L
S ::= L $
S ::= L, i$
L ::= L$
L ::= i$

L ::= L
L ::= L
L ::= L

L ::= i
L ::= L
L ::= L

S ::= L
S ::= L

L ::= L
L ::= L

L ::= L
L ::= L

L ::= i
L ::= i
L ::= i
Step 6: Reduce L

S ::= L $
L ::= L , i
L ::= i

S ::= L $
L ::= L , i

S ::= L $

L ::= L , i
Step 7: Shift 2 (,)
Step 8: Shift 2 (,)

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

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

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

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

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

CIS 461/561, Spring 2001
Step 9: Shift 5 (i)

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

CIS 461/561, Spring 2001
Step 10: Reduce L

1. S ::= •L$
   L ::= •L, i
   L ::= •i$

2. S ::= L•$
   L ::= L•, i$

3. S ::= L$
   L ::= L
   L ::= L, i$

4. L ::= i•$

5. L ::= L•, i$

6. L ::= L, i•$

\[ a, b, c \]

CIS 461/561, Spring 2001
Step 11: Shift 1 (L)
Step 12: Shift 2 ($)

S ::= • L $

L ::= • L , i
L ::= • i

S ::= L •$

L ::= L , i
L ::= L , i

S ::= L$

L ::= i
L ::= L ,
L ::= L , i

L$

i,

L$
Step 13: Reduce $S$

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

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

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

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

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

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

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

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

Consider:

\[ S ::= E \]

\[ E ::= E + E \]

\[ E ::= i \]

(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

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

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

\[ (S) ::= (E) \$ \]
\[ (E) ::= (E) + (T) \]
\[ (E) ::= T \]
\[ (T) ::= (T) \ast i \]
\[ (T) ::= i \]

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

LR(0) items construction plus

- Lookahead of start symbol is $\$
- Lookahead propagates except for closure
- In closure, we pick up the following symbol

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

\[
\langle \text{decl} \rangle \quad ::= \quad \langle \text{func-decl} \rangle \mid \langle \text{var-decl} \rangle \\
\langle \text{var-decl} \rangle \quad ::= \quad \langle \text{var-type} \rangle \langle \text{var-list} \rangle \mid ; \\
\langle \text{var-type} \rangle \quad ::= \quad \text{id} \\
\langle \text{var-list} \rangle \quad ::= \quad \langle \text{var-list} \rangle , \text{id} \mid \text{id} \\
\langle \text{func-decl} \rangle \quad ::= \quad \langle \text{func-type} \rangle \langle \text{func-name} \rangle \\
\quad \quad \quad \quad \left( \langle \text{args} \rangle \right) \mid ; \\
\langle \text{func-type} \rangle \quad ::= \quad \text{id} \mid \text{void} \\
\langle \text{func-name} \rangle \quad ::= \quad \text{id}
\]

(Build cfsm on chalkboard; requires more than one token of lookahead. Fix grammar to use ;, (, and , 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.
A grammar that requires LALR(1) lookahead

\[
\begin{align*}
\langle P \rangle &::= \langle Stmt \rangle \; \$ \\
\langle Stmt \rangle &::= \langle Decl \rangle \; ; \\
\langle Stmt \rangle &::= \langle Assign \rangle \; ; \\
\langle Decl \rangle &::= \langle Type \rangle \; \langle Var \rangle \\
\langle Assign \rangle &::= \langle Var \rangle \; = \; \langle Expr \rangle \\
\langle Var \rangle &::= * \; ident \\
\langle Var \rangle &::= ident \\
\langle Expr \rangle &::= \langle Var \rangle \\
\langle Expr \rangle &::= \langle Expr \rangle \; * \; \langle Var \rangle
\end{align*}
\]
Yacc/Bison form of grammar

%left ’*’
%token IDENT
%

Stmt:       Decl ’;’
          | Assign ’;’
          ;

Decl:       Typename Variable
          ;

Assign:     Variable ’=’ Expression
          ;

Typename:   IDENT
          ;

Variable:   ’*’ IDENT
          | IDENT
          ;

Expression: Variable
          | Expression ’*’ Variable
          ;

CIS 461/561, Spring 2001
### Syntax Rules

<table>
<thead>
<tr>
<th>Production</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>P ::= @ Stmt $</td>
<td>Program starts with Stmt followed by a <code>$</code> symbol.</td>
</tr>
<tr>
<td>S ::= @ Decl ; S ::= @ Assign ;</td>
<td>Statement is a sequence of Decl and Assign statements.</td>
</tr>
<tr>
<td>Decl ::= @ Type Var</td>
<td>Declaration is a Type followed by a Var.</td>
</tr>
<tr>
<td>Type ::= @ ident</td>
<td>Type is an ident.</td>
</tr>
<tr>
<td>Assign ::= @ Var = Expr</td>
<td>Assign is a Var followed by an <code>=</code> and then an Expr.</td>
</tr>
<tr>
<td>Var ::= @ * ident</td>
<td>Var can be an asterisk followed by an ident.</td>
</tr>
<tr>
<td>Var ::= @ ident</td>
<td>Var can also be an ident.</td>
</tr>
</tbody>
</table>

### Shift-Reduce Table

<table>
<thead>
<tr>
<th>State</th>
<th>Action</th>
<th>Symbol</th>
<th>Next State</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Shift 6</td>
<td>$</td>
<td>6</td>
</tr>
<tr>
<td>0</td>
<td>Error</td>
<td>;</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>Error</td>
<td>*</td>
<td>-</td>
</tr>
<tr>
<td>6</td>
<td>Shift 7</td>
<td>=</td>
<td>-</td>
</tr>
</tbody>
</table>

- `shift 6` indicates a shift action.
- `error` indicates an error state.
- `;` and `*` are symbols read from the input.
- `=` indicates an equality.
- `;` and `*` are symbols read from the input.
- `=` indicates an equality.

CIS 461/561, Spring 2001