**Outline**

- Review of shift-reduce parsers
- Limitations of LR(0) grammars
- SLR, LR(1), LALR parsers
- parser generators

---

**Administration**

- Programming Assignment 1 due now!
- Homework 2 due in 1 week
- Programming Assignment 2 due in 2 + ε weeks

---

**Bottom-up parsing**

- Apply productions backwards as reductions
- Builds parse tree from terminal symbols up towards start symbol
- Shift-reduce parser constructs right-most derivation using sequence of shift and reduce operations

---

**Shift-reduce parsing**

- Parsing is a sequence of shifts and reduces
- **Shift** -- push look-ahead token onto stack
  - stack input
  - \( (1+2+(3+4))+5 \) shift 1
  - \( (1+2+(3+4))+5 \)
- **Reduce** -- Replace symbols \( \gamma \) in top of stack with non-terminal symbol \( X \), corresponding to production \( X \rightarrow \gamma \) (pop \( \gamma \), push \( X \))
  - stack input
  - \( \frac{S}{E \rightarrow id} \) reduce \( S \rightarrow S+E \)
  - \( \frac{S}{E \rightarrow (S)} \) reduce \( S \rightarrow S+E \)
  - \( \frac{E}{(3+4)+5} \) reduce \( S \rightarrow S+E \)
  - \( \frac{S}{E \rightarrow id} \)

---

**LR(0) states**

- Problem: when to reduce & what production?
- Idea: state summarizes parser stack prefix, keeps track of what productions might need to be the next reduce
  - what tokens can be shifted
  - when a production is complete
- A state is a set of items: partially completed productions that might appear in derivation
An LR(0) grammar: lists

\[
\begin{align*}
S & \rightarrow \text{id} \mid \text{(} \underline{L} \text{)} \\
L & \rightarrow \text{L}, \text{S} \mid \text{S}
\end{align*}
\]

\[
(x, (y, z), w)
\]

Constructing states

\[
\begin{align*}
S & \rightarrow (\text{L}) \\
L & \rightarrow \cdot, \text{S} \mid \text{S}
\end{align*}
\]

Full DFA (Appel p. 63)

<table>
<thead>
<tr>
<th>State</th>
<th>Input</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>S</td>
<td>x</td>
<td>shift</td>
</tr>
<tr>
<td>S</td>
<td>y</td>
<td>goto 8</td>
</tr>
<tr>
<td>S</td>
<td>z</td>
<td>reduce</td>
</tr>
</tbody>
</table>

Determining current state

- Run parser stack through the DFA
- State tells us what productions might be reduced next

<table>
<thead>
<tr>
<th>Stack</th>
<th>Input</th>
<th>State</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>(L, x)</td>
<td>y</td>
<td>state = 6</td>
<td>?</td>
</tr>
</tbody>
</table>

Optimization

- Attach parser state to each stack entry
- Don’t have to traverse DFA all over again after each reduction
- Just start from state at top of stack when RHS γ is removed, take single step for LHS non-terminal.

<table>
<thead>
<tr>
<th>Derivation</th>
<th>Stack</th>
<th>Input</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>(x, y)</td>
<td>\text{id}</td>
<td>(x, y)</td>
<td>shift, goto 3</td>
</tr>
<tr>
<td>(x, y)</td>
<td>\text{L} \text{L}</td>
<td>(x, y)</td>
<td>shift, goto 2</td>
</tr>
<tr>
<td>(x, y)</td>
<td>\text{L} \text{L}</td>
<td>\text{y}</td>
<td>reduce S→id</td>
</tr>
<tr>
<td>(S, y)</td>
<td>\text{L}</td>
<td>S, y</td>
<td>reduce L→S</td>
</tr>
<tr>
<td>(L, y)</td>
<td>\text{L}</td>
<td>\text{y}</td>
<td>shift, goto 8</td>
</tr>
<tr>
<td>(L, y)</td>
<td>\text{L}</td>
<td>\text{y}</td>
<td>reduce S→id</td>
</tr>
<tr>
<td>(L, y)</td>
<td>\text{L}</td>
<td>\text{y}</td>
<td>reduce L→S</td>
</tr>
<tr>
<td>(L, y)</td>
<td>\text{L}</td>
<td>\text{y}</td>
<td>shift, goto 9</td>
</tr>
<tr>
<td>(L, y)</td>
<td>\text{L}</td>
<td>\text{y}</td>
<td>reduce S→id</td>
</tr>
<tr>
<td>(L, y)</td>
<td>\text{L}</td>
<td>\text{y}</td>
<td>reduce L→S, S</td>
</tr>
<tr>
<td>(L)</td>
<td>\text{L}</td>
<td>\text{y}</td>
<td>shift, goto 6</td>
</tr>
<tr>
<td>(L)</td>
<td>\text{L}</td>
<td>\text{y}</td>
<td>reduce S→(L)</td>
</tr>
</tbody>
</table>

Stack entries labeled w/state

<table>
<thead>
<tr>
<th>State</th>
<th>Input</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>S</td>
<td>\text{L}</td>
<td>\text{S}</td>
</tr>
</tbody>
</table>
Implementation: LR parsing table

<table>
<thead>
<tr>
<th>Terminal symbols</th>
<th>Non-terminal symbols</th>
</tr>
</thead>
<tbody>
<tr>
<td>Next action</td>
<td>Next state</td>
</tr>
</tbody>
</table>

- **Action table**: Used at every step to decide whether to shift or reduce.
- **Goto table**: Used only when reducing, to determine next state.

Shift-reduce parser table

- **Actions in table**
  1. If state action is shift and goto state $n$:
     - Shift and goto state $n$.
  2. If state action is reduce using $X \rightarrow \gamma$:
     - Pop symbols $\gamma$ off stack.
     - Using state label of top (end) of stack, look up $X$ in goto table and goto that state.

- DFA + Stack = Push-down Automaton (PDA)

List grammar parse table

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
</table>
| () | id | , | $S$ | $L$
| 1 | s3 | s2 |   | g4 |
| 2 | S→id | S→id | S→id | S→id |
| 3 | s3 | s2 | accept | g7 |
| 4 |   |   |   | g5 |
| 5 | S→(L)S→(L)S→(L)S→(L)S→(L) |   |   |   |
| 6 | L→S | L→S | L→S | L→S |
| 7 | s3 | s2 |   | g9 |
| 8 | L→L,S | L→L,S | L→L,S | L→L,S |
| 9 |   |   |   |   |

List grammar parse table

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
</table>
| () | id | , | $S$ | $L$
| 1 | s3 | s2 |   | 4 |
| 2 | S→id | S→id | S→id | S→id |
| 3 | s3 | s2 | accept | 7 |
| 4 |   |   |   | 5 |
| 5 | S→(L)S→(L)S→(L)S→(L)S→(L) |   |   |   |
| 6 | L→S | L→S | L→S | L→S |
| 7 | s3 | s2 |   | 9 |
| 8 | L→L,S | L→L,S | L→L,S | L→L,S |
| 9 |   |   |   |   |

LR(0) Limitations

- An LR(0) machine only works if states with reduce actions have a single reduce action -- in those states, always reduce ignoring input.
- With more complex grammar, construction gives states with shift/reduce or reduce/reduce conflicts.
- Need to use look-ahead to choose.

How about sum grammar?

- $S \rightarrow S + E | E$
- $E \rightarrow \text{num} | ( S )$

- This is LR(0)
- Right-associative version isn’t:

  - $S \rightarrow E + S | E$
  - $E \rightarrow \text{num} | ( S )$
LR(0) construction

$$S \rightarrow E + S \\ E \rightarrow num \ | \ (S)$$

SLR grammars

- Idea: Only add reduce action to table if look-ahead symbol is in the FOLLOW set of the non-terminal being reduced
- Eliminates some conflicts
- $FOLLOW(S) = \{ \$, \)$
- Many language grammars are SLR

SLR(1) parsing

- Gets as much power as possible out of 1 look-ahead symbol
- LR(1) grammar = recognizable by a shift/reduce parser with 1 look-ahead.
- LR(1) items keep track of look-ahead symbol expected to follow this production

LR(0): $S \rightarrow .S + E$
LR(1): $S \rightarrow .S + E +$

LR(1) construction

$S \rightarrow E + S \ | \ E$
$E \rightarrow num \ | \ (S)$

LALR grammars

- Problem with LR(1): too many states
- LALR(1) (Look-Ahead LR)
  - Merge any two LR(1) states whose items are identical except look-ahead
  - Results in smaller parser tables -- works extremely well in practice

Classification of Grammars
How are parsers written?

- Automatic parser generators: yacc, bison, CUP
- Accept LALR(1) grammar specification
- *plus*: declarations of precedence, associativity

**Associativity**

\[
\begin{align*}
S &\rightarrow S + E \mid E \\
E &\rightarrow \text{num} \mid (S)
\end{align*}
\]

\[
E \rightarrow E + E \mid \text{num} \mid (E)
\]

What happens if we run this grammar through LALR construction?

**Conflict!**

\[
E \rightarrow E + E \mid \text{num} \mid (E)
\]

\[
\begin{align*}
E &\rightarrow E + E \mid E + E \\
E &\rightarrow E + E + E
\end{align*}
\]

shift/reduce conflict

1+2+3

\[
\begin{align*}
\text{shift: } 1\cdot(2+3) \\
\text{reduce: } (1+2)+3
\end{align*}
\]

**Grammar in CUP**

non terminal E: terminal PLUS, LPAREN...

precedence left PLUS:

"When shifting + conflicts with reducing a production containing +, choose reduce"

\[
E ::= E \PLUS E
\]

| LPAREN E RPAREN |
| NUMBER |

**Precedence**

- Also can handle operator precedence

\[
\begin{align*}
E &\rightarrow E + E \mid T \\
T &\rightarrow T\times T \mid \text{num} \mid (E)
\end{align*}
\]

\[
\begin{align*}
E &\rightarrow E + E \mid E\times E \\
&\mid \text{num} \mid (E)
\end{align*}
\]

**Conflicts w/o precedence**

\[
\begin{align*}
E &\rightarrow E + E \mid E\times E \\
&\mid \text{num} \mid (E)
\end{align*}
\]

\[
\begin{align*}
E &\rightarrow E \PLUS E \\
E &\rightarrow E \times E
\end{align*}
\]

\[
\begin{align*}
E &\rightarrow E \PLUS E \mid E\times E \mid \text{any}
\end{align*}
\]
Precedence in CUP
precedence left PLUS;
precedence left TIMES; // TIMES > PLUS
E ::= E PLUS E | E TIMES E | ...

\[
\begin{align*}
E \rightarrow E \cdot E & \quad \text{any} \\
E \rightarrow E \times E & \quad \text{+} \\
E \rightarrow E + E & \quad \times \\
E \rightarrow E \cdot E & \quad \text{any}
\end{align*}
\]

Rule: in conflict, choose reduce if production symbol higher precedence than shifted symbol; choose shift if vice-versa

Summary
- Look-ahead information makes SLR(1), LALR(1), LR(1) grammars expressive
- Automatic parser generators support LALR(1)
- Precedence, associativity declarations simplify grammar writing
- Can we use parsers for programs other than compilers?