Parsing -3

- Deterministic table-driven parsing techniques
  - Pictorial view of TD and BU parsing
  - BU (shift-reduce) Parsing
    - Handle, viable prefix, items, closures, goto's
    - LR(k): SLR(1), LR(1)
    - Problems with SLR
    - LALR(k) an optimization
  - Using ambiguity to an advantage

Aho, Sethi, Ullman, *Compilers: Principles, Techniques and Tools*
Aho + Ullman, *Theory of Parsing and Compiling, vol II.*

---

A View During TD Parsing

S derives a γ, a string of terminals; X is nonterminal at top of stack, X derives γ; Initially X==S, a == e, γ is input

---

A View During TD Parsing

S derives a γ, a string of terminals; X is nonterminal at top of stack, X derives γ; Initially X==S, a == e, γ is input

---

A View During TD Parsing

S derives a γ, a string of terminals; X is nonterminal at top of stack, X derives γ; Initially X==S, a == e, γ is input

---

A View During TD Parsing

S derives a γ, a string of terminals; X is nonterminal at top of stack, X derives γ; Initially X==S, a == e, γ is input

---

A View During TD Parsing

S derives a γ, a string of terminals; X is nonterminal at top of stack, X derives γ; Initially X==S, a == e, γ is input

---

A View During TDParsing

S derives a γ, a string of terminals; X is nonterminal at top of stack, X derives γ; Initially X==S, a == e, γ is input

---

A View During TD Parsing

S derives a γ, a string of terminals; X is nonterminal at top of stack, X derives γ; Initially X==S, a == e, γ is input
A View During BU Parsing

\[ S \rightarrow^* \alpha A w \rightarrow^* \alpha \beta w, \]

so \( \beta \) is the handle.

Intuitive Comparison

LR(\(k\)) can recognize \( A \rightarrow \alpha \) knowing \( w, x \), and \( \text{First}_A(z) \).
LL(\(k\)) can recognize \( A \rightarrow \alpha \) knowing only \( w \) and \( \text{First}_A(x) \).
Therefore, the set of languages recognizable by LR(\(k\)) contain those recognizable by LL(\(k\)).
**BU Parsing (Shift-Reduce)**

Handle - part of sentential form last added in a rightmost derivation. 
BU parsing is “handle hunting”

Rightmost derivation of \( a+b+c \), handles in red:

\[
\begin{align*}
S & \rightarrow E \\
E & \rightarrow E + T \\
E & \rightarrow T \\
T & \rightarrow id
\end{align*}
\]

(1) \( S \rightarrow E \)  
(2) \( E \rightarrow E + T \)  
(3) \( E \rightarrow T \)  
(4) \( T \rightarrow id \)

---

**Shift-Reduce Parser, Example**

Actions: shift, reduce, accept, error

<table>
<thead>
<tr>
<th>Stack</th>
<th>Input</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>$</td>
<td>$</td>
<td></td>
</tr>
<tr>
<td>$ id1</td>
<td>+ id2 + id3</td>
<td>shift</td>
</tr>
<tr>
<td>$ T</td>
<td>+ id2 + id3</td>
<td>reduce (3)</td>
</tr>
<tr>
<td>$ E</td>
<td>+ id2 + id3</td>
<td>shift</td>
</tr>
<tr>
<td>$ E + id2</td>
<td>+ id3</td>
<td>shift</td>
</tr>
<tr>
<td>$ E + T</td>
<td>+ id3</td>
<td>reduce (2)</td>
</tr>
<tr>
<td>$ E</td>
<td>+ id3</td>
<td>shift</td>
</tr>
<tr>
<td>$ E + id3</td>
<td>$</td>
<td>reduce (4)</td>
</tr>
<tr>
<td>$ E + T</td>
<td>$</td>
<td>reduce (2)</td>
</tr>
<tr>
<td>$ E</td>
<td>$</td>
<td>reduce (1)</td>
</tr>
<tr>
<td>$ S</td>
<td>$</td>
<td>accept</td>
</tr>
</tbody>
</table>

(1) \( S \rightarrow E \)  
(2) \( E \rightarrow E + T \)  
(3) \( E \rightarrow T \)  
(4) \( T \rightarrow id \)
Problems

Shift-reduce conflicts
\[ S \to \text{if E then S} \mid \text{if E then S else S} \mid \text{other} \]
On stack: if E then S
Input: else
Should shift trying for 2nd alternative or reduce by first rule?

Reduce-reduce conflicts
if \( A \to \alpha \) and \( B \to \alpha \) both in grammar
When \( \alpha \) on stack, how do we know which production to choose?

Predictive Parsing

- Top Down: LL(k), Bottom Up: LR(k)
- Avoids backtracking while parsing by using lookahead into input
- NO cases where more than 1 action possible
- LR parsing algorithms developed in the mid-1970s: powerful enablers of table-driven compilation
LR(k)

- Left to right scan parsing does a rightmost derivation in reverse, using $k$ symbols of lookahead into input
- Examples
  - Simple LR - SLR(1)
    - Cheap but doesn’t always work
  - LR(k)
    - Most powerful and most expensive
- All SLR(1) languages are also LR(1), but parsers generated by corresponding grammars for the same language will differ in size.
- LR(k) catches syntax errors as early as possible in a left-to-right scan of the input and works for most modern PLs

LR Parsing

- FSA is embedded in parser which is a Pushdown automaton
- $(\text{top}_{\text{stack}}, \text{input}\_\text{symbol})$ accesses a particular entry in the parser table
  - Shift to state $s$
  - Reduce by $A \rightarrow \beta$
  - Accept
  - Error
- $\text{Goto}: (\text{state}, \text{top}_{\text{stack}}) \rightarrow \text{state}$
LR Parsing

- **Idea:** continue to stack inputs until have **handle** on top of stack and then reduce to its non-terminal symbol
- **Viable prefix** - set of prefixes of right sentential forms which can appear on a stack of a shift/reduce parser
- **Goto** function is transition function of DFA that recognizes viable prefixes of the grammar
- **Idea** is that while a viable prefix is on top of the stack the goto function continues the parse towards getting a handle on top of the stack that can be reduced
Building an SLR Parser

- Need states, goto’s, Follow sets
- Item - rule with embedded dot
  \[ S \rightarrow \cdot E \]
- Closure of item I
  \[ I \cup \{ B \rightarrow \cdot \gamma , \text{ if } A \rightarrow \alpha . B \beta \text{ in I} \} \]
- States built from items and their closures

SLR(1) Example - States

<table>
<thead>
<tr>
<th>I_0: \ S \rightarrow \ E \</th>
<th>\ E \rightarrow \cdot \ E + T \</th>
<th>\ E \rightarrow \ T \</th>
<th>\ T \rightarrow \cdot \ id</th>
</tr>
</thead>
<tbody>
<tr>
<td>I_1: \ S \rightarrow \cdot E \</td>
<td>\ E \rightarrow \cdot E + T</td>
<td></td>
<td></td>
</tr>
<tr>
<td>I_2: \ E \rightarrow \cdot T</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>I_3: \ T \rightarrow \cdot \ id</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>I_4: \ E \rightarrow \cdot E + T</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>I_5: \ E \rightarrow \cdot E + T</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Closure of \( S \rightarrow \cdot E \)
Example - Goto’s + Follow sets

goto (0, E) = 1  \quad \text{goto (0, id) = 3}
goto (0, T) = 2  \quad \text{goto (1, +) = 4}
goto (4, T) = 5  \quad \text{goto (4, id) = 3}

goto \{\text{set of items}, X\} =
\text{closure \{[A \rightarrow \alpha X \beta] | [A \rightarrow \alpha X \beta] \text{ in \{set of items\}}\}}
\text{where } X \text{ is a terminal or nonterminal}

Follow(S) = \{$\}$
Follow(E) = Follow(T) = \{ +, $\}$

Rules for forming Follow Sets

ASU p 189

1. Follow(S) contains $\$
2. If $A \rightarrow \alpha X \beta$ then everything in First(\beta) except $\epsilon$, is put into Follow(X)
3. If $A \rightarrow \alpha X$ or $A \rightarrow \alpha X \beta$ where First(\beta) contains $\epsilon$, then Follow(A) is contained in Follow(X)
Example - Parser Table

si, shift to state I; r(j) reduce by rule j

States\ inputs:  \ goto’s

<table>
<thead>
<tr>
<th></th>
<th>id</th>
<th>$</th>
<th>E</th>
<th>T</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>s3</td>
<td>1</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>s4</td>
<td>accept</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>r(3)</td>
<td>r(3)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>r(4)</td>
<td>r(4)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>s3</td>
<td>5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>r(2)</td>
<td>r(2)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Example

Stack | input | action
-----|-------|--------
0 | id1 + id2 $ | s3
0 | id1 3 | + id2 $ | r(4), goto on T
0 | T 2 | + id2 $ | r(3), goto on E
0 | E 1 | + id2 $ | s4
0 | E 1 + 4 | id2 $ | s3
0 | E 1 + 4 id2 3 | $ | r(4), goto on T
0 | E 1 + 4 T 5 | $ | r(2), goto on E
0 | E 1 | $ | accept
SLR(1) Parser Rules

• If $A \rightarrow \alpha \cdot a \beta$ is in state $I_j$ and $\text{goto}(I_j, a)$ is $I_r$ then $(I_j, a)$ transitions by shift $r$ (sr)
• If $A \rightarrow \alpha$ is in state $I_j$, set action [j,a] to reduce $A \rightarrow \alpha$ for all $a$ in $\text{Follow}(A)$
  – Note: $A \neq S$
• If $S \rightarrow E \cdot$ in $I_j$, action (j,$\cdot$) is accept
• Any table entry not defined is error.

Problems

• Shift-reduce conflicts happen when $Ab$ can occur in some sentential form and $b \in \text{Follow}(A)$.

\begin{align*}
  S \rightarrow & L = R \\
  S \rightarrow & R \\
  L \rightarrow & * R \\
  L \rightarrow & id \\
  R \rightarrow & L \\
\end{align*}

$I_0$: 
\begin{align*}
  S & \rightarrow . L = R \\
  S & \rightarrow . R \\
  R & \rightarrow . L \\
  L & \rightarrow . * R \\
  L & \rightarrow . id \\
\end{align*}

$I_1$: 
\begin{align*}
  S & \rightarrow L = R \ (1) \\
  R & \rightarrow L \ (2) \\
\end{align*}

In state $I_1$ 1st choice: shift when see $=$ in input(item 1);
2nd choice: reduce on $=$ because $=$ in $\text{Follow}(R)$ (item 2);
Note: $S \rightarrow L = R \rightarrow * R = R$ ..., but this is not a rightmost derivation!
Problems, cont.

Can see that a rightmost derivation is:
\[ S \rightarrow L = R \rightarrow L = L \rightarrow L = id \rightarrow \ast R = id \rightarrow \ast L = \ast id = id \]

Therefore, should reduce \( \ast R \) to \( L \) when see \( = \),
not shift in order to get \( \ast R \) onto the stack.

**Problem is that we can’t distinguish those**

**Follow elements corresponding to a**

**rightmost derivation in a specific context.**

---

Nomenclature in ASU

- An item \([ A \rightarrow \beta . \gamma ]\) is **valid** for **viable prefix** \( \alpha \beta \) if \( S \xrightarrow{\text{rm}} \alpha A w \xrightarrow{\text{rm}} \alpha \beta \gamma w \).
  - Means can continue towards accumulating an handle on the stack by shifting
  - Previously, shift would have changed viable prefix \( \ast R \) to nonviable prefix \( \ast R = \)

- If \( I \) is set of items valid for **viable prefix** \( \beta \) then goto(I, X) is set of items valid for **viable prefix** \( \beta X \) where \( X \) is terminal or nonterminal
LR(1) Parsing

- LR items include a lookahead symbol, (into the input) which helps in conflict resolution.
- Need new closure rule:
  - For \([ A \rightarrow \alpha \cdot B \gamma, a ]\) item add \([B \rightarrow \delta, b]\) for every \(b\) in \(\text{First}(\gamma a)\).

Example

\[ I_0: \quad S \rightarrow . E, $ \quad \text{- initial item} \]
\[ E \rightarrow . E + T, $ \quad \text{- closure initial item} \]
\[ E \rightarrow . T, $ \]
\[ E \rightarrow . E + T, + \quad \text{- closure 1st red item} \]
\[ E \rightarrow . T, + \]
\[ T \rightarrow . id, $ \quad \text{- closure 2nd red item} \]
\[ T \rightarrow . id, + \quad \text{- closure 2nd blue item} \]

Will write these in more compact form by combining lookaheads.

For \([ A \rightarrow a \cdot B \gamma, a ]\) item add \([B \rightarrow . d, b]\)
for every \(b\) in \(\text{First}(\gamma a)\).
Example, LR(1) Parser

\begin{align*}
I_0: & S \rightarrow .E, \$ & I_1: & \text{[goto (} I_0 , E) ] \\
E \rightarrow & .E + T, \$/+ & S \rightarrow & E ., \$ \\
E \rightarrow & . T, \$/+ & S \rightarrow & E . + T, \$/+ \\
T \rightarrow & .id , +/$ & I_2: & \text{[goto (} I_0 , T) ] \\
I_4: & \text{[goto (} I_1 , +\) ]} \\
E \rightarrow & E + . T, \$/+ & I_3: & \text{[goto (} I_0 , id) ] \\
T \rightarrow & . id, \$/+ & T \rightarrow & id ., \$/+ \\
I_5: & \text{[goto (} I_4 , T) ]} \\
E \rightarrow & E + T ., \$/+ \\
\end{align*}

LR(1) Parser

- Reduce based on lookaheads in item which are a subset of Follow set
- Rules similar to SLR(1)
  - Shift in \( I_k , [A \rightarrow \alpha . a \beta , b] \), goto (\( I_k , a \)) = \( I_j \)
  - Reduce \([A \rightarrow \alpha . , b]\) reduce \( \alpha \) to \( A \) on \( b \)
  - Accept \([S \rightarrow E ., \$], \) accept on \( \$ \)
LALR Parsing

• **Idea:** merge all states with common first components in their LR(1) items

• Implementation problem: need to reduce number of states to get smaller parser table

• Reduced size parser will perform
  – Same as LR on correct inputs (will be parsed by LALR)
  – On incorrect inputs, LR may find error faster; LALR will never do an incorrect shift but may do some wrong reductions

LALR Parsing

• Conceptually, build LALR(1) parser from LR(1) parser
  – Merge all states with same first components
  – Union all goto’s of these merged states (goto’s are independent of second components)

• Correctness of conceptual derivation
  – Can never produce a *shift-reduce conflict* or else \([A \rightarrow \alpha . \; a] \text{ and } [B \rightarrow \beta . \; a \; \gamma , \; b] \) existed in some LR(1) state
Useful Ambiguous Grammars

• Used to build compact parse trees
  – Get rid of useless nonterminal to nonterminal productions (e.g., $S \rightarrow E \rightarrow T$)

• Conflicts resolvable through desired properties of operators (e.g., precedence)

• Generate smaller parsers
  – Example of expression grammar

Example

\[
\begin{align*}
S & \rightarrow E \\
E & \rightarrow E + E \\
E & \rightarrow id \\
I_0 & : goto (I_0, E) \\
I_1 & : goto (I_0, E) \\
I_2 & : goto (I_1, +) \\
I_3 & : goto (I_2, E) \\
I_4 & : goto (I_0, id) \\
E & \rightarrow E + . E \\
E & \rightarrow E + E \\
E & \rightarrow E . + E \\
E & \rightarrow . id \\
\end{align*}
\]

Choose reduce action making + left associative; can resolve operator precedence clashes the same way (e.g., + versus *)
Grammar Classification

Context-freeLanguages

\{0^n 1^n | n \geq 1\} union \{0^n 1^{2n} | n \geq 1\}

LR(k) \sim LR(1)

LL(k)

SLR(k)

LALR(k)

S \rightarrow L = R | R
L \rightarrow *R | a
R \rightarrow L