Parsing Wrapup

Filling in the LR(1) ACTION and GOTO Tables

The algorithm

1. Set \( S_\text{init} = S \)
2. If \( i \) is a terminal and \( \text{goto}(x, a) = x_n \Rightarrow \text{accept} \)
3. Else if \( i \) is a non-terminal and \( \text{ ACTION } (x, S) \leftarrow \text{ "shift $T" \} \) (shift if lookahead \( = a \))
4. Else if \( i \) is a non-terminal and \( \text{ ACTION } (x, S) \leftarrow \text{ "reduce $A \Rightarrow a" \} \) (shift if lookahead \( = a \))
5. If \( \text{ goto } (x, n) = x \)
6. Then \( \text{ goto } (x, n) \leftarrow k \) (store transitions for nonterminals)

Many items generate no table entry (the rhs' for $S$ are in $S_n$)

e.g., \( [A \Rightarrow B, a] \) does not, but closure ensures that all

Roadmap (Where are we?)

Last lecture
- Shift-reduce parser
- LR(1) parsing
  - LR(1) item
  - Computing goto
  - LR(1) canonical collection

This lecture
- LR(1) parsing
  - Building ACTION / GOTO tables
  - Shift/reduce and reduce/reduce conflicts
  - SLR(1), LALR(1), operator precedence
  - Error recovery

Example - Building LR(1) ACTION and GOTO Table
Example - Building LR(1) Action and Goto Table

| P0 | S ::= E |
| P1 | E ::= T + E |
| P2 | T ::= id |

**ACTION**

- S: \(\text{id} \rightarrow \text{B} E T\)
- P: \(\text{id} \rightarrow \text{E} T\)
- S: \(\text{id} \rightarrow \text{E} T\)
- S: \(\text{id} \rightarrow \text{E} T\)

**GOTO**

- S to T: \(\text{id} \rightarrow \text{E} T\)
- P to S: \(\text{id} \rightarrow \text{E} T\)
- S to P: \(\text{id} \rightarrow \text{E} T\)

What can go wrong?

- If set \(S\) contains \(\{A \rightarrow \text{id} \cdot \text{E}\} \text{ and } \{\text{E} \rightarrow \text{id} \cdot \text{E}\}\)
  - First item generates "shift", second generates "reduce"
  - Both define \(\text{ACTION}(S, E)\) — cannot do both actions
  - This is a fundamental ambiguity, called a **shift/reduce conflict**

- If set \(S\) contains \(\{A \rightarrow \text{id} \cdot \text{E}\} \text{ and } \{\text{E} \rightarrow \text{id} \cdot \text{E}\}\)
  - Each generates "reduce", but with a different production
  - Both define \(\text{ACTION}(S, E)\) — cannot do both reductions
  - This fundamental ambiguity is called a **reduce/reduce conflict**

In either case, the grammar is not LR(1)

**Solutions**

- Modify grammar to eliminate conflict
- Specify how conflict should be resolved
  - Can be used to assign precedence & associativity (to operators)

Shift/reduce Error Example

- **Grammar**
  
  \[
  \begin{align*}
  P1: & \quad S \rightarrow E \\
  P2: & \quad E \rightarrow E + E \\
  P3: & \quad \text{a} \\
  \end{align*}
  \]

- **State in canonical collection of LR(1) items**
  
  \[
  [E \rightarrow E + E \cdot \{a\}]
  [E \rightarrow E + E \cdot \{a\}]
  \]

- **Entry in ACTION / GOTO table for lookahead +**
  
  \[
  \begin{align*}
  \text{Shift 3:} & \quad \text{GO TO table for lookahead +} \\
  \text{Reduce P2:} & \quad (E \rightarrow E + E)
  \end{align*}
  \]

Reduce/reduce Error Example

- **Grammar**
  
  \[
  \begin{align*}
  P1: & \quad S \rightarrow E \\
  P2: & \quad E \rightarrow A \\
  P3: & \quad \text{a} \\
  P4: & \quad A \rightarrow a \\
  \end{align*}
  \]

- **State in canonical collection of LR(1) items**
  
  \[
  [E \rightarrow a \cdot \{a\}]
  [A \rightarrow a \cdot \{a\}]
  \]

- **Entry in ACTION / GOTO table for lookahead $**
  
  \[
  \begin{align*}
  \text{Reduce P3 (E \rightarrow a)} \\
  \text{Reduce P4 (A \rightarrow a)}
  \end{align*}
  \]

Left Recursion versus Right Recursion

- **Right recursion**
  - Required for termination in top-down parsers
  - Produces right-associative operators

- **Left recursion**
  - Works fine in bottom-up parsers
  - Limits required stack space
  - Produces left-associative operators

- **Rule of thumb**
  - Left recursion for bottom-up parsers
  - Right recursion for top-down parsers

Associativity

- **Normally defined by programming language**
- **What difference does it make?**
- **Can change answers in floating-point arithmetic**
- **Exposes a different set of common subexpressions**

- **Consider x + y + z**

- **Ideal operator**

- **Right association**

- **Left association**

- **What if x + y + z occurs elsewhere? Or x + y or x + z?**
- **What if x = 2 & z = 17?**
  - Neither left nor right exposes 19
  - I.e., optimizer cannot replace x + z with 19 in output code
Resolving Conflicts

- Precedence and associativity may be used to resolve conflicts in ambiguous grammars
- When comparing operator on stack with lookahead
  - Shift if lookahead has
    - Higher precedence
    - Same precedence, right associative
  - Reduce if lookahead has
    - Lower precedence
    - Same precedence, left associative
- Advantage
  - Can use smaller (ambiguous) grammars
  - Example
    - $E \rightarrow E \cdot E | E \cdot E | E / E | E \cdot E | id | num$

Operator Precedence Grammars

- Alternative approach to shift-reduce parsing
- Given a sentential form $aABb$, three possibilities
  1. $A$ in handle, $B$ not in handle
     - $A \rightarrow aBb$ is reduced before $B$
     - $A > B$
  2. $A$ and $B$ both in handle
     - $A \rightarrow aBb$ is reduced at the same time
     - $A = B$
  3. $B$ in handle, $A$ not in handle
     - $B \rightarrow aBb$ is reduced before $A$
     - $B > A$
- Handle is composed of
  - $< s < s, > s > s, < s \cdot s, > s \cdot s, \ldots$, etc...
- To decide whether to shift or reduce, compare top of stack with lookahead (ignoring nonterminals)
  - Shift if $< s$ or $>$
  - Reduce if $>$ (left end of handle is closest to top of stack)

Operator Precedence Grammar Example

<table>
<thead>
<tr>
<th>The Grammar</th>
<th>Stack</th>
<th>Input</th>
<th>Precedence</th>
</tr>
</thead>
<tbody>
<tr>
<td>$E \rightarrow E \cdot E</td>
<td>E \cdot E</td>
<td>E / E</td>
<td>E \cdot E</td>
</tr>
<tr>
<td>$+$</td>
<td>$&gt; &lt; &lt; &lt;$</td>
<td>$1d + 1d \cdot 1d</td>
<td>1d &gt; 1d</td>
</tr>
<tr>
<td>$*$</td>
<td>$&gt; &lt; &lt; &lt;$</td>
<td>$1d + 1d \cdot 1d</td>
<td>1d &gt; 1d</td>
</tr>
<tr>
<td>$1d$</td>
<td>$&gt; &lt; &lt; &lt;$</td>
<td>$1d + 1d \cdot 1d</td>
<td>1d &gt; 1d</td>
</tr>
<tr>
<td>$S$</td>
<td>$&gt; &lt; &lt; &lt;$</td>
<td>$1d + 1d \cdot 1d</td>
<td>1d &gt; 1d</td>
</tr>
</tbody>
</table>

Shrinking the ACTION/GOTO Tables

Three options:
- Combine terminals such as number & identifier, $\cdot$ & $\cdot A$
  - Directly removes a column, may remove a row
  - For expression grammar, 198 (vs. 384) table entries
- Combine rows or columns
  - Implement identical rows once & remap states
  - Requires extra indirection on each lookup
  - Use separate mapping for ACTION & for GOTO
- Use another construction algorithm
  - Both SLR(1) and LR(1) produce smaller tables
  - Implementations are readily available

SLR(1) Parser

- Build ACTION / GOTO table using LR(0) items
  - Problem - when to perform reduction for $A \rightarrow \cdot B$?
  - Solution - reduce $A \rightarrow \cdot B$ only when lookahead $\in$ FOLLOW(A)
- Algorithm

  - $\forall$ set $s \in S$
    - $i \in \{A, B, \cdot, \cdot, X, Y, Z\}$ and $g(a)(s, i) = s_0 \rightarrow T$
      - to left of terminal a
      - then $\text{ACTION}(s, a) = \text{"shift"} k$
      - // shift if lookahead = a
    - else if $i \in \{S, =, -$)
      - // start production done,
      - then $\text{ACTION}(s, i) = \text{"accept"}$
      - // accept if lookahead = $s$
    - else if $i \in \{A, B, \cdot, \cdot \}$ and $a \in$ FOLLOW(A)
      - // all the way to right
      - then $\text{ACTION}(s, a) = \text{"reduce A-$B"}$
      - // reduce if lookahead
    - $\forall$ $a \in NP$
      - // is in FOLLOW(A)

- if $g(a)(s, i) = s_0$
  - then $\text{GOTO}(s, i) = k$
  - // store transitions for nonterminals

Example - SLR(1) Parser

<table>
<thead>
<tr>
<th>ACTION</th>
<th>GOTO</th>
</tr>
</thead>
<tbody>
<tr>
<td>$S_0 \rightarrow E \cdot T$</td>
<td>$S_1 \rightarrow T$</td>
</tr>
<tr>
<td>$E \rightarrow T \cdot E$</td>
<td>$S_2 \rightarrow T$</td>
</tr>
<tr>
<td>$E \rightarrow E \cdot T$</td>
<td>$S_3 \rightarrow T$</td>
</tr>
<tr>
<td>$T \rightarrow 4$</td>
<td>$S_4 \rightarrow 1d$</td>
</tr>
<tr>
<td>$S_7 \rightarrow 1d$</td>
<td>$S_8 \rightarrow 1d$</td>
</tr>
<tr>
<td>$E \rightarrow E \cdot T$</td>
<td>$S_9 \rightarrow 1d$</td>
</tr>
</tbody>
</table>
### LALR(1) Parser

- Core of set of LR(1) items
  - Set of LR(0) items derived by ignoring lookahead symbols
  - Example
    - LR(0) state
    - LALR(1) parser
    - Merge two sets of LR(1) items (states), if same core
    - Result
      - Potentially much smaller set of states
      - Some as SLR(1) parser
      - May introduce reduce/reduce conflicts
      - Will not introduce shift/reduce conflicts

### LR(k) versus LL(k) (Top-down Recursive Descent)

**Finding Reductions**
- LR(k) ⇒ Each reduction in the parse is detectable with
  - Complete left context (everything to left of handle)
  - Reducible phrase (handle) itself
  - k terminal to right of handle
- LL(k) ⇒ Parser must select the next rule based on
  - Complete left context (everything up to \( k \) including NT to expand)
  - \( k \) terminals to right of nonterminal to expand

**Thus, LR(k) is more powerful since it examines more context**

For example, consider grammar \( S \rightarrow a | \epsilon \)

- LL(1)
  - No, cannot decide which production with lookahead = \( a \)
- LR(1)
  - Yes, can shift \( ab \) or \( a \) onto stack before deciding on reduction

### LR(k) Parsers

- Properties
  - More powerful than LL(k) parsers
  - Most general non-backtracking shift-reduce parser
  - Detects errors as soon as possible in left-to-right scan of input
  - Contents of stack are viable prefixes
    - Possible for remaining input to lead to successful parse

### LR(k) versus LL(k) Summary

<table>
<thead>
<tr>
<th>Advantages</th>
<th>Disadvantages</th>
</tr>
</thead>
<tbody>
<tr>
<td>Top-down recursive descent</td>
<td>Hand-coded</td>
</tr>
<tr>
<td>Fast</td>
<td>High maintenance</td>
</tr>
<tr>
<td>Good locality</td>
<td>Right associativity</td>
</tr>
<tr>
<td>Good error detection</td>
<td></td>
</tr>
<tr>
<td>LR(1)</td>
<td>Large working sets</td>
</tr>
<tr>
<td>Fast</td>
<td>Poor error messages</td>
</tr>
<tr>
<td>Deterministic</td>
<td>Large table sizes</td>
</tr>
<tr>
<td>Left automaton</td>
<td></td>
</tr>
<tr>
<td>Left associativity</td>
<td></td>
</tr>
</tbody>
</table>

### LR(0) versus SLR(1) versus LR(1)

**Example grammar**

\[
S \rightarrow S \\
S \rightarrow S, a | a
\]

- LR(0) ?
- LR(1) ?
- SLR(1) ?
LALR(1) versus LR(1)

Example grammar

```
S → S
S → aAd | bBd | aBe | bAe
A → c
B → c
```

LR(0)?

LR(1)?

LALR(1)?

Hierarchy of Context-Free Languages

Context-free languages

- LR(0)
- LR(1)
- LL(0) languages
- Simple precedence languages
- LR(4) = LR(1)
- Deterministic languages (LR(0))
- LL(1) languages
- Operator precedence languages

The inclusion hierarchy for context-free languages:

Hierarchy of Context-Free Grammars

- Context-free grammars
  - Floyd-Evans
  - Parsable
  - Unambiguous
  - CFGs
  - Operator
  - Precedence

- LR(4)
- LR(1)
- LALR(1)
- SLR(1)
- LL(1)
- LL(0)

* Operator precedence includes some ambiguous grammars
* LL(1) is a subset of SLR(1)

Error Recovery in Shift-Reduce Parsers

The problem: parser encounters an invalid token

Goal: Want to parse the rest of the file

Basic idea (panic mode):

- Assume something went wrong while trying to find handle for nonterminal A
- Pretend handle for A has been found: pop "handle", skip over input to find terminal that can follow A

Restarting the parser (panic mode):

- Find a restartable state on the stack (has transition for nonterminal A)
- Move to a consistent place in the input (token that can follow A)
- Perform (error) reduction (for nonterminal A)
- Print an informative message

Error Recovery in YACC (ASU p.264)

Yacc’s (bison’s) error mechanism (note: version dependent!)

- Designated token `error`
- Used in error productions of the form A → `error` α
- α specifies synchronization points

When error is discovered

- Pops stack until it finds state where it can shift the `error` token
- Resumes parsing to match α

- Special cases:
  - a = w, where w is string of terminals: skip input until w has been read
  - a = e: skip input until state transition on input token is defined
- Error productions can have actions

Error Recovery in YACC

```
cmpdstmt: BEG stmt_list END
stmt_list : stmt
| stmt_list ";" stmt
| error ( "**Error: illegal statement\n")
```