Bottom-up parsers

Properties

• start with input string, end with start symbol
• apply productions in reverse to input, replacing right-hand side (rhs) of production with lhs nonterminal
• final result is a rightmost derivation, in reverse.
• bottom-up parsers can use current stack and lookahead to choose production
• one type of bottom-up parsers called LR(k) are more powerful than LL(k) parsers because they can see the entire rhs before choosing a production

Definitions

• the handle is defined as the combination of
  1) rhs to be replaced, and 2) its position
• replacement step is called a reduction

Bottom-up parsing example

Consider the grammar

1 S ::= a A B e
2 A ::= A b c
3   | b
4 B ::= d

and the input string abbcde.

<table>
<thead>
<tr>
<th>Production</th>
<th>Sentential Form</th>
<th>Handle</th>
</tr>
</thead>
<tbody>
<tr>
<td>—</td>
<td>abbcde</td>
<td>3,2 (A ⇐ b)</td>
</tr>
<tr>
<td>3</td>
<td>aAbcde</td>
<td>2,4 (A ⇐ A b c)</td>
</tr>
<tr>
<td>2</td>
<td>aAde</td>
<td>4,3 (B ⇐ d)</td>
</tr>
<tr>
<td>4</td>
<td>aABe</td>
<td>1,4 (S ⇐ a A B e)</td>
</tr>
<tr>
<td>1</td>
<td>S</td>
<td>—</td>
</tr>
</tbody>
</table>

The problem is deciding when and which rhs to reduce.
Bottom-up parsing using tables

A table-driven bottom-up parser looks like

Stack two items per state: state and symbol
Table building tools are readily available (yacc, CUP)

Shift-reduce parsing

Shift-reduce parsers

- one approach to bottom-up parsing
- are simple to understand
- have a simple, table-driven, shift-reduce skeleton
- encode grammatical knowledge in tables

A shift-reduce parser has just four canonical actions:

1. shift — next input symbol is shifted onto the top of the stack
2. reduce — right end of handle is on top of stack;
   locate left end of handle within the stack;
   pop handle off stack and push appropriate non-terminal lhs
3. accept — terminate parsing and signal success
4. error — call an error recovery routine
### Skeleton parser

```python
code
push s_0
token = next_token()
repeat forever
    s = top of stack
    if action[s,token] = "shift s_i" then
        push token
        push s_i
        token = next_token()
    else if action[s,token] = "reduce A ::= \beta" then
        pop 2 * | \beta | symbols
        s = top of stack
        push A
        push goto[s, A]
    else if action[s, token] = "accept" then
        return
    else error()
```

This takes \( k \) shifts, \( l \) reduces, and 1 accept, where \( k \) is the length of the input string and \( l \) is the length of the reverse rightmost derivation.

### Example parser

<table>
<thead>
<tr>
<th>ACTION</th>
<th>GOTO</th>
</tr>
</thead>
<tbody>
<tr>
<td>id</td>
<td>+</td>
</tr>
<tr>
<td>P0 S ::= E</td>
<td>S_0</td>
</tr>
<tr>
<td>P1 E ::= T + E</td>
<td>S_1</td>
</tr>
<tr>
<td>P2 T ::= id</td>
<td>S_2</td>
</tr>
<tr>
<td>P3 T ::= id</td>
<td>S_3</td>
</tr>
<tr>
<td></td>
<td>S_4</td>
</tr>
<tr>
<td></td>
<td>S_5</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Stack</th>
<th>Input</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>$ 0</td>
<td>id + id $</td>
<td>shift 3</td>
</tr>
<tr>
<td>$ 0 id 3</td>
<td>+ id $</td>
<td>reduce P3 (T ::= id)</td>
</tr>
<tr>
<td>$ 0 T 2</td>
<td>+ id $</td>
<td>shift 4</td>
</tr>
<tr>
<td>$ 0 T 2 + 4</td>
<td>id $</td>
<td>shift 3</td>
</tr>
<tr>
<td>$ 0 T 2 + 4 id 3</td>
<td>$</td>
<td>reduce P3 (T ::= id)</td>
</tr>
<tr>
<td>$ 0 T 2 + 4 T 2</td>
<td>$</td>
<td>reduce P2 (E ::= T)</td>
</tr>
<tr>
<td>$ 0 T 2 + 4 E 5</td>
<td>$</td>
<td>reduce P1 (E ::= T + E)</td>
</tr>
<tr>
<td>$ 0 E 1</td>
<td>$</td>
<td>accept</td>
</tr>
</tbody>
</table>
LR(1) grammars

Informally, we say that a grammar $G$ is LR(1) if we can find the sequence of handles for a reverse rightmost derivation using at most 1 token of lookahead past the end of the handle.

Properties

- virtually all context-free programming language constructs can be expressed in an LR(1) form
- LR grammars are the most general grammars that can be parsed by a non-backtracking, shift-reduce parser
- efficient shift-reduce parsers can be implemented for LR(1) grammars
- LR parsers detect an error as soon as possible in a left-to-right scan of the input
- LR grammars describe a proper superset of the languages recognized by LL (predictive) parsers

Shift-reduce parsing

Definitions

- a right-sentential form is any string that may occur in a legal rightmost derivation
- a viable prefix of a right-sentential form is any prefix that does not continue past the right end of its rightmost handle

Shift-reduce parsers

- operator precedence define precedence between operands to guide reductions
- $\text{SLR}(1) = \text{LR}(0) + \text{FOLLOW}$
  construct DFA for recognizing viable prefix, use FOLLOW to guide reductions
- $\text{LR}(1)$ — construct DFA for recognizing viable prefix, storing lookahead information in DFA
- $\text{LALR}(1)$ — construct DFA for recognizing viable prefix, propagating lookahead information in DFA
LR(0) items

An LR(0) item is a string \([\alpha]\), where
\[\alpha\] is a production from \(G\) with a \(\bullet\) at some position in the rhs

The \(\bullet\) indicates how much of an item we have seen at a given state in the parse.

\([A ::= \bullet XY Z]\) indicates that the parser is looking for a string that can be derived from \(XYZ\)

\([A ::= XY \bullet Z]\) indicates that the parser has seen a string derived from \(XY\) and is looking for one derivable from \(Z\)

**LR(0) Items**

\(A ::= XY Z\) generates 4 LR(0) items.

1. \([A ::= \bullet XY Z]\)
2. \([A ::= X \bullet Y Z]\)
3. \([A ::= XY \bullet Z]\)
4. \([A ::= XYZ\bullet]\)

**LR(0) machine**

Definitions

- closure of \([A ::= \alpha \bullet B\beta]\) contains itself and any items of form \([B ::= \bullet \gamma]\), repeat for new items.
- goto(\(X\)) of \([A ::= \alpha \bullet X\beta]\) contains the closure of \([A ::= \alpha X \bullet \beta]\).

**LR(0) DFA construction**

1. begin with closure of start symbol \([S ::= \bullet \alpha]\)
2. for each state, calculate goto(\(X\)) for all grammar symbols \(X\), generating states
3. repeat step 2 for all newly generated states

**Properties**

- states in the DFA are sets of LR(0) items
- states represent viable prefixes of productions
- to recognize viable prefixes of language, save state of current production on stack when reducing new nonterminal
LR(0) items

The Grammar

\[
\begin{align*}
P1 & : E ::= T + E \\
P2 & : | T \\
P3 & : T ::= \text{id}
\end{align*}
\]

The Augmented Grammar

\[
\begin{align*}
P0 & : S' ::= E \\
P1 & : E ::= T + E \\
P2 & : | T \\
P3 & : T ::= \text{id}
\end{align*}
\]

Example LR(0) states

\[
\begin{align*}
S_0: & \quad [S' ::= \bullet E], \\
& \quad [E ::= \bullet T + E], \\
& \quad [E ::= \bullet T], \\
& \quad [T ::= \bullet \text{id}] \\
S_1: & \quad [S' ::= E \bullet] \\
S_2: & \quad [E ::= T \bullet + E], \\
& \quad [E ::= T \bullet] \\
S_3: & \quad [T ::= \text{id} \bullet] \\
S_4: & \quad [E ::= T + \bullet E], \\
& \quad [E ::= \bullet T + E], \\
& \quad [E ::= \bullet T], \\
& \quad [T ::= \bullet \text{id}] \\
S_5: & \quad [E ::= T + E \bullet]
\end{align*}
\]
**LR(1) items**

We can build SLR parsers using LR(0) items and FOLLOW information. But, we can get more powerful parsers by keeping track of lookahead information in the states of the LR parser.

An *LR(k)* item is a pair \([\alpha, \beta]\), where

- \(\alpha\) is a production from \(G\) with a \(\bullet\) at some position in the rhs
- \(\beta\) is a lookahead string containing \(k\) symbols (terminals or \texttt{eof})

**LR(1) items**

- example: \([A ::= X \bullet YZ, a]\)
- several LR(1) items may have the same core
  \[[A ::= X \bullet YZ, a]\]
  \[[A ::= X \bullet YZ, b]\]
  we represent this as
  \[[A ::= X \bullet YZ, \{a, b\}]\]

---

**LR(1) lookahead**

What’s the point of all these lookahead symbols?

- carry them along to allow us to choose correct reduction when there is any choice
- lookahead are bookkeeping, unless item has \(\bullet\) at right end.
  - in \([A ::= X \bullet YZ, a]\), \(a\) has no direct use
  - in \([A ::= XYZ\bullet, a]\), \(a\) is useful
- allows use of (non-invertible) grammars where productions have the same rhs

The point

For \([A ::= a\bullet, a]\) and \([B ::= a\bullet, b]\), we can decide between reducing to \(A\) and to \(B\) by looking at limited right context!
**LR(1) machine**

Definitions

- **closure** of \([A ::= \alpha \bullet B \beta, a]\) contains itself and any items of form 
  \([B ::= \bullet \gamma, \text{FIRST}(\beta a)],\) repeat for new items.
- **goto** \((X)\) of \([A ::= \alpha \bullet X \beta, a]\) contains the closure of \([A ::= \alpha X \bullet \beta, a].\)

**LR(1) DFA construction**

1. begin w/ closure of start symbol \([S ::= \bullet \alpha, \text{eof}]\)
2. for each state, calculate **goto** \((X)\) for all grammar symbols \(X\), generating states
3. repeat step 2 for all newly generated states

**Properties**

- \([A \rightarrow X \bullet YZ, \alpha] \Rightarrow \text{have recognized } X \& YZ \text{ would be valid}\)
- \([A \rightarrow X \bullet YZ, \alpha] \Rightarrow [Y \rightarrow \bullet \beta, \gamma] \& [Y \rightarrow \bullet \delta, \eta] \text{ are also valid, where } \gamma, \eta \in \text{FIRST}(Z\alpha)\)
- recognizing \(Y\) takes parser to \([A \rightarrow XY \bullet Z, \alpha]\)

---

**Example LR(1) states**

\[S_0\]: \([S' ::= \bullet E, \$]\) \hspace{2cm} \([S_3\]: \([T ::= \text{id } \bullet, +]\) \hspace{2cm} \([S_5\]: \([E ::= T + E, \$]\)

\[E ::= T \bullet + E, \$]\) \hspace{2cm} \([T ::= \text{id } \bullet, \$]\)

\[E ::= \bullet T, \$]\) \hspace{2cm} \([T ::= \text{id }, +]\)

\[T ::= \text{id }, \$]\)

\[S_1\]: \([S' ::= E \bullet, \$]\) \hspace{2cm} \([S_3\]: \([T ::= \text{id } \bullet, +]\)

\[S_2\]: \([E ::= T \bullet + E, \$]\) \hspace{2cm} \([S_4\]: \([E ::= T + E, \$]\)

\[E ::= \bullet T, \$]\)

\[T ::= \text{id }, +]\)

\[T ::= \text{id }, \$]\)

CMSC 430 Lecture 5, Page 15
**LR(1) table construction**

The Algorithm

1. if $S$ appears on rhs of production, create augmented grammar $G'$ by adding $S' := S$
2. construct the collection of sets of LR(1) items for $G'$.
3. State $i$ of the parser is constructed from $I_i$.
   (a) if $[A ::= \alpha \bullet a \beta, b] \in I_i$ and $\text{goto}(I_i, a) = I_j$, then set $\text{action}[i, a]$ to “shift $j$”. (a must be a terminal)
   (b) if $[A ::= \alpha \bullet, a] \in I_i$, then set $\text{action}[i, a]$ to “reduce $A ::= \alpha$”.
   (c) if $[S' ::= S \bullet, \text{eof}] \in I_i$, then set $\text{action}[i, \text{eof}]$ to “accept”.
4. If $\text{goto}(I_i, A) = I_j$, then set $\text{goto}[i, A]$ to $j$.
5. All other entries in action and goto are set to “error”
6. The initial state of the parser is the state constructed from the set containing the item $[S' ::= \bullet S, \text{eof}]$.

Example ACTION and GOTO tables

The Augmented Grammar

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>P0</td>
<td>$S' := E$</td>
<td></td>
</tr>
<tr>
<td>P1</td>
<td>$E ::= T + E$</td>
<td></td>
</tr>
<tr>
<td>P2</td>
<td>$T := id$</td>
<td></td>
</tr>
<tr>
<td>P3</td>
<td>$T := id$</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>ACTION</th>
<th>GOTO</th>
</tr>
</thead>
<tbody>
<tr>
<td>id</td>
<td>shift 3</td>
<td>1 2</td>
</tr>
<tr>
<td>+</td>
<td></td>
<td>E T</td>
</tr>
<tr>
<td>$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$</td>
<td>accept</td>
<td></td>
</tr>
<tr>
<td>$</td>
<td>shift 4</td>
<td>reduce P2</td>
</tr>
<tr>
<td>$</td>
<td>reduce P3</td>
<td>reduce P3</td>
</tr>
<tr>
<td>$</td>
<td>shift 3</td>
<td>5 2</td>
</tr>
<tr>
<td>$</td>
<td>reduce P1</td>
<td></td>
</tr>
</tbody>
</table>

The "reduce" actions are determined by the lookahead entries in the LR(1) items
What can go wrong?

Multiply actions may exist in the ACTION table.

Two cases arise

shift/reduce

This is called a shift/reduce conflict. In general, it indicates an ambiguous construct in the grammar.

- can modify the grammar to eliminate it
- can resolve in favor of shifting

classic example: dangling else

reduce/reduce

This is called a reduce/reduce conflict. Again, it indicates an ambiguous construct in the grammar.

- often, no simple resolution
- parse a nearby language

classic example: PL/I call and subscript

Resolving conflicts

Precedence and associativity can be used to resolve shift/reduce conflicts in ambiguous grammars.

- same precedence & right associative, or higher precedence ⇒ shift
- same precedence & left associative, or lower precedence ⇒ reduce

Advantages:

- more concise, albeit ambiguous, grammars
- shallower parse trees ⇒ fewer reductions

⇒ a simpler expression grammar

\[
<\text{expr}> ::= <\text{expr}> \ast <\text{expr}>
\mid <\text{expr}> \div <\text{expr}>
\mid <\text{expr}> + <\text{expr}>
\mid <\text{expr}> - <\text{expr}>
\mid ( <\text{expr}> )
\mid -<\text{expr}>
\mid \text{id}
\mid \text{num}
\]
Operator precedence parsers

Another approach to shift-reduce parsing is to use operator precedence. Given $S \Rightarrow^* \alpha S_1 S_2 \beta$, there are three possible precedence relations between $S_1$ and $S_2$.

1. $S_1$ in handle, $S_2$ not  
   ($S_1$ reduced before $S_2$)  
   $S_1 > S_2$
2. both in handle  
   (reduced at same time)  
   $S_1 = S_2$
3. $S_2$ in handle, $S_1$ not  
   ($S_2$ reduced before $S_1$)  
   $S_1 < S_2$

A handle is thus composed of: 
$<>, < = >, < == >, ...$

To decide whether to shift or reduce, compare top of stack with lookahead (ignoring nonterminals):

- Shift if $<$ or $=$
- Reduce if $>$

Left end of handle is marked by first $<$ found

CMSC 430

Parsing example

The Grammar

$E ::= E + E \mid E * E \mid id$

|   | + | * | id | $ $
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>$&gt;$</td>
<td>$&lt;$</td>
<td>$&lt;$</td>
</tr>
<tr>
<td>*</td>
<td>$&gt;$</td>
<td>$&gt;$</td>
<td>$&lt;$</td>
</tr>
<tr>
<td>id</td>
<td>$&gt;$</td>
<td>$&gt;$</td>
<td>$&gt;$</td>
</tr>
<tr>
<td>$ $</td>
<td>$&lt;$</td>
<td>$&lt;$</td>
<td>$&lt;$</td>
</tr>
</tbody>
</table>

Stack | Input | Precedence |
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>$ $</td>
<td>id + id * id $ $</td>
<td>$ $ &lt; id</td>
</tr>
<tr>
<td>$ &lt; id$</td>
<td>+ id * id $ $</td>
<td>id &gt; + $ $</td>
</tr>
<tr>
<td>$ &lt; E$</td>
<td>+ id * id $ $</td>
<td>$ $ &lt; + $ $</td>
</tr>
<tr>
<td>$ &lt; E +$</td>
<td>id * id $ $</td>
<td>+ &lt; id $ $</td>
</tr>
<tr>
<td>$ &lt; E + &lt; id$</td>
<td>* id $ $</td>
<td>id &gt; * $ $</td>
</tr>
<tr>
<td>$ &lt; E + &lt; E$</td>
<td>* id $ $</td>
<td>+ &lt; * $ $</td>
</tr>
<tr>
<td>$ &lt; E + &lt; E *$</td>
<td>id $ $</td>
<td>* &lt; id $ $</td>
</tr>
<tr>
<td>$ &lt; E + &lt; E * &lt; id$</td>
<td>id $ $</td>
<td>id &gt; $ $</td>
</tr>
<tr>
<td>$ &lt; E + &lt; E * &lt; id$</td>
<td>id $ $</td>
<td>$ $ &gt;$ $</td>
</tr>
<tr>
<td>$ &lt; E + E$</td>
<td>+ id $ $</td>
<td>$ + &gt;$ $</td>
</tr>
<tr>
<td>$ &lt; E$</td>
<td>$ $</td>
<td>$ $ &gt;$ $</td>
</tr>
</tbody>
</table>

CMSC 430
Error recovery in shift-reduce parsers

The problem

- encounter an invalid token
- bad pieces of tree on stack

We want to *parse* the rest of the file

Restarting the parser

- find a restartable state on the stack
- move to a consistent place in the input
- print an informative message *(line number)*

Yacc’s error mechanism

- designated token *error*
- valid in any production
- when an error is discovered, pops the stack until *error* is legal

Error recovery example

```plaintext
stmt_list : stmt
    | stmt_list ; stmt
```

can be augmented with *error*

```plaintext
stmt_list : stmt
    | error
    | stmt_list ; stmt
```

this should

- throw out the erroneous statement
- synchronize at “;” or “end”
- invoke `yyerror("syntax error")`

Other “natural” places for errors

- all the “lists”
- missing parentheses or brackets
- extra operator or missing operator
**LR(1) grammars**

Informally, we say that a grammar $G$ is LR(1) if we can find the sequence of handles for a reverse rightmost derivation using at most 1 token of lookahead past the end of the handle.

**Properties**

- virtually all context-free programming language constructs can be expressed in an LR(1) form
- LR grammars are the most general grammars that can be parsed by a non-backtracking, shift-reduce parser
- efficient shift-reduce parsers can be implemented for LR(1) grammars
- LR parsers detect an error as soon as possible in a left-to-right scan of the input
- LR grammars describe a proper superset of the languages recognized by LL (predictive) parsers

**LALR(1) parsers**

Problem

- LR(1) parsers are powerful, but have many more states than LR(0) (approximately $\times 10$ for Pascal)
- larger state tables longer to construct, run

**LALR(1) parsers**

- define the core of a set of LR(1) items to be the set of LR(0) items derived by ignoring the lookahead symbols.
- example of two sets of LR(1) items with same core:
  - $\{[A \Rightarrow \alpha \cdot \beta, a], [A \Rightarrow \alpha \cdot \beta, b]\}$, and
  - $\{[A \Rightarrow \alpha \cdot \beta, c], [A \Rightarrow \alpha \cdot \beta, d]\}$
- if two sets of LR(1) items, $I_i$ and $I_j$, have the same core, we can merge the states that represent them in the ACTION and GOTO tables
- almost as powerful as LR(1), same size as LR(0)
**LALR(1) parsers**

There are two approaches to constructing LALR(1) parsing tables

1. build LR(1) sets of items, then merge states with same core
2. build LR(0) sets of items, then propagate lookahead information.

**LALR(1) properties**

- LALR(1) parsers have same number of states as LR(0) parsers (core LR(0) items are the same)
- may perform *reduce* rather than *error*, but will catch error before more input is processed
- LALR derived from LR with no shift-reduce conflict will also have no shift-reduce conflict
- LALR may create reduce-reduce conflict not in LR from which LALR is derived
- used by utilities such as *yacc, bison, cup*

---

**LR(k) languages**

```
SLR(1) LALR(1) LR(1) LR(k)
Grammars
SLR(1) LALR(1) LR(1) LR(k)
Languages
```
Recursive Descent A hand coded recursive descent parser directly encodes a grammar (typically an LL(1) grammar) into a series of mutually recursive procedures. It has most of the linguistic limitations of LL(1).

LL(k) An LL(k) parser must be able to recognize the use of a production after seeing only the first $k$ symbols of its right hand side.

Operator Precedence An ad hoc shift-reduce parser suitable for small expression grammars.

LR(k) An LR(k) parser must be able to recognize the occurrence of the right hand side of a production after having seen all that is derived from that right hand side with $k$ symbols of lookahead.

<table>
<thead>
<tr>
<th></th>
<th>Advantages</th>
<th>Disadvantages</th>
</tr>
</thead>
<tbody>
<tr>
<td>recursive descent LL(1)</td>
<td>fast</td>
<td>hand-coded</td>
</tr>
<tr>
<td></td>
<td>simple</td>
<td>maintenance hard</td>
</tr>
<tr>
<td></td>
<td>automatable</td>
<td>no left recursion</td>
</tr>
<tr>
<td></td>
<td>good error recovery</td>
<td>LL(1) ⊂ LR(1)</td>
</tr>
<tr>
<td>operator precedence</td>
<td>fast</td>
<td>poor error detection</td>
</tr>
<tr>
<td></td>
<td>simple</td>
<td>L(G) ≠ L(parser)</td>
</tr>
<tr>
<td></td>
<td>small table</td>
<td>only small languages</td>
</tr>
<tr>
<td>LR(1)</td>
<td>fast</td>
<td>larger table size</td>
</tr>
<tr>
<td></td>
<td>early error detection</td>
<td>poor error recovery</td>
</tr>
<tr>
<td></td>
<td>automatable</td>
<td></td>
</tr>
</tbody>
</table>