This Lecture

• Parsing strategies: top-down and bottom-up.
• Shift-Reduce parsing theory.
• LR(0) parsing.
• LR(0), LR(k), and LALR(k) grammars

Parsing Strategies

There are two basic strategies for parsing:

• **Top-down parsing**:
  - Attempts to construct the parse tree *from the root downward*.
  - Traces out a *leftmost derivation*.
  - E.g. Recursive-Descent Parsing (see G52MAL).

• **Bottom-up parsing**:
  - Attempts to construct the parse tree *from the leaves working up toward the root*.
  - Traces out a *rightmost derivation in reverse*.

Top-Down: Leftmost Derivation

Consider the grammar:

\[ S \rightarrow aABe \quad A \rightarrow bc.A \mid c \quad B \rightarrow d \]

Call sequence for predictive parser on *abcde*:

\[
\begin{align*}
\text{parseS} & \Rightarrow \text{lm} aABe \\
\text{read a} & \Rightarrow \text{lm} abcABe \\
\text{read b} & \Rightarrow \text{lm} abccBe \\
\text{read c} & \Rightarrow \text{lm} abccde \\
\text{parseB} & \Rightarrow \text{lm} abcde \\
\text{read d} & \\
\text{read e} &
\end{align*}
\]
Shift-Reduce Parsing

**Shift-reduce parsing** is a general style of bottom-up syntax analysis:
- Works from the leaves toward the root of the parse tree.
- Has two basic actions:
  - **Shift** (read) next terminal symbol.
  - **Reduce** a sequence of read terminals and previously reduced nonterminals corresponding to the RHS of a production to LHS nonterminal of that production.

Shift-Reduce Parsing: Idea

How can we know when and what to reduce???

Idea:
- Construct a DFA where each state is labelled by “all possibilities” given the input and reductions thus far. (Similar to how an NFA is turned into a DFA.)
- Whenever reduction is possible, if there is only one possible reduction, then it is always clear what to do.

Will make this more precise in the following.

Bottom-Up: Rightmost Der. in Reverse

Consider (again) the grammar:

\[
S \rightarrow aABe \quad A \rightarrow bcA \mid c \quad B \rightarrow d
\]

Reduction steps for the sentence \(abcde\) to \(S\)

- \(abcde\) (reduce by \(A \rightarrow c\))
- \(abcAde\) (reduce by \(A \rightarrow bcA\))
- \(aAde\) (reduce by \(B \rightarrow d\))
- \(aABe\) (reduce by \(S \rightarrow aABe\))
- \(S\)

Trace out rightmost derivation in reverse:

\[
S \Rightarrow aABe \Rightarrow aAde \Rightarrow abcAde \Rightarrow abccde
\]

How can we know when and what to reduce???

LL, LR, and LALR parsing (1)

Three important classes of parsing methods:

- **LL(\(k\)):**
  - input scanned *Left* to right
  - *Left*most derivation
  - \(k\) symbols of lookahead

- **LR(\(k\)):**
  - input scanned *Left* to right
  - *Right*most derivation in reverse
  - \(k\) symbols of lookahead

- **LALR(\(k\)):** Look*Ahead* LR, simplified LR parsing
LL, LR, and LALR parsing (2)

By extension, the classes of grammars these methods can handle are also classified as LL($k$), LR($k$), and LALR($k$).

Why study LR and LALR parsing?

- These methods handle a wide class of grammars of practical significance.
- In particular, handles left- and right-recursive grammars (but left rec. needs less stack).
- LALR is a good compromise between expressiveness and space cost of implementation.
- Consequently, many parser generator tools based on LALR.
- We will mainly study LR(0) parsing because it is the simplest, yet uses the same fundamental principles as LR(1) and LALR(1).

Shift-Reduce Parsing Theory (1)

Some terminology:
- An item for a CFG is a production with a dot anywhere in the RHS.

For example, the items for the grammar

\[
S \to aAc \\
A \to Ab | \epsilon
\]

are

\[
S \to \cdot aAc \\
A \to \cdot Ab \\
S \to a \cdot Ac \\
A \to A \cdot b \\
S \to aA \cdot c \\
A \to Ab \cdot \\
S \to aAc \cdot \\
A \to \\
\]

Shift-Reduce Parsing Theory (2)

- Recap: Given a CFG $G = (N, T, P, S)$, a string $\phi \in (N \cup T)^*$ is a sentential form for $G$ iff $S \xrightarrow{G}^* \phi$.
- A right-sentential form is a sentential form that can be derived by a rightmost derivation.
- A handle of a right-sentential form $\phi$ is a substring $\alpha$ of $\phi$ such that $S \xrightarrow{G}^* \delta Aw \Rightarrow^* \delta \alpha w$ and $\delta \alpha w = \phi$, where $\alpha, \delta, \phi \in (N \cup T)^*$, and $w \in T^*$.
For example, consider the grammar:

\[ S \rightarrow aABe \quad A \rightarrow bcA \mid c \quad B \rightarrow d \]

The following is a rightmost derivation:

\[ S \Rightarrow_{rm} aABe \Rightarrow_{rm} aAde \Rightarrow_{rm} abcAde \]

\(aABe, aAde\) and \(abcAde\) are right-sentential forms. Handle for each? \(aABe, d, \) and \(bcA\)

For an unambiguous grammar, the rightmost derivation is unique. Thus we can talk about "the handle" rather than merely "a handle".

A viable prefix of a right-sentential form \(\phi\) is any prefix \(\gamma\) of \(\phi\) ending no farther right than the right end of the handle of \(\phi\).

An item \(A \rightarrow \alpha \cdot \beta\) is valid for a viable prefix \(\gamma\) if there is a rightmost derivation

\[ S \Rightarrow_{rm} \delta Aw \Rightarrow_{rm} \delta\alpha\beta w \]

and \(\delta\alpha = \gamma\).

An item is complete if the the dot is the rightmost symbol in the item.

Consider the grammar

\[ S \rightarrow aABe \quad A \rightarrow bcA \mid c \quad B \rightarrow d \]

and the rightmost derivation

\[ S \Rightarrow_{rm} aABe \Rightarrow_{rm} aAde \Rightarrow_{rm} abcAde \]

The right-sentential form \(abcAde\) has handle \(bcA\).

Viable prefixes? \(\epsilon, a, ab, abc, abcA\).

Last derivation step \(aAde \Rightarrow abcAde\) by production \(A \rightarrow bcA\), meaning the handle is \(bcA\).

Valid item for each non-\(\epsilon\) viable prefix of \(abcAde\) considering this particular derivation only?

Viable prefix | Valid item
-------------|-----------------
    \(a\)     | \(A \rightarrow \cdot bcA\)
    \(ab\)    | \(A \rightarrow b \cdot cA\)
    \(abc\)   | \(A \rightarrow bc \cdot A\)
    \(abcA\)  | \(A \rightarrow bcA \cdot \)

Any complete valid item?
Shift-Reduce Parsing Theory (7)

Knowing the valid items for a viable prefix allows a rightmost derivation in reverse to be found:

- If \( A \rightarrow \alpha \cdot \) is a complete valid item for a viable prefix \( \gamma = \delta \alpha \) of a right-sentential form \( \gamma \in T^* \), then it appears that \( A \rightarrow \alpha \) can be used at the last step, and that the previous right-sentential form is \( \delta Aw \).
- If this indeed always is the case for a CFG \( G \), then for any \( x \in L(G) \), since \( x \) is a right-sentential form, previous right-sentential forms can be determined until \( S \) is reached, giving a right-most derivation of \( x \).

LR(0) Parsing (1)

- A CFG for which knowing a complete valid item is enough to determine the previous right-sentential form is called LR(0).
- For every CFG whatsoever, the set of viable prefixes is regular.
- Thus, an efficient parser can be developed for an LR(0) CFG based on a DFA for recognising viable prefixes and their valid items.
- The states of the DFA are sets of items valid for a recognised viable prefix.

Shift-Reduce Parsing Theory (8)

Of course, if \( A \rightarrow \alpha \cdot \) is a complete valid item for a viable prefix \( \gamma = \delta \alpha \), in general, we only know it may be possible to use \( A \rightarrow \alpha \) to derive \( \gamma w \) from \( \delta Aw \). For example:

- \( A \rightarrow \alpha \cdot \) may be valid because of a different rightmost derivation \( S \Rightarrow_{\text{rm}} \delta Aw \Rightarrow_{\text{rm}} \phi w' \).
- There could be two or more complete items valid for \( \gamma \).
- There could be a handle of \( \gamma w \) that includes symbols of \( w \).

LR(0) Parsing (2)

A DFA recognising viable prefixes for the CFG

\[
\begin{align*}
S & \rightarrow aABe \\
A & \rightarrow bcA \mid c \\
B & \rightarrow d
\end{align*}
\]
LR(0) Parsing (3)

Drawing conventions for “LR DFAs”:

- For the purpose of recognizing the set of viable prefixes, all drawn states are considered accepting.
- Error transitions and error states are not drawn.

LR(0) Parsing (4)

How to construct such a DFA is beyond the scope of this course. See e.g. Aho, Sethi, Ullman (1986) for details. However, some observations:

- Recall that the viable prefixes for the right-sentential form $abcAde$ are $\epsilon, a, ab, abc, abcA$. They are indeed all recognised by the DFA (all states are considered accepting).
- Recall that the item $A \rightarrow bc \cdot A$ is valid for the viable prefix $abc$. The corresponding DFA state indeed contains that item. (Along with more items in this case!)

LR(0) Parsing (5)

- Recall that item $A \rightarrow bcA \cdot$ is a complete valid item for the viable prefix $abcA$. The corresponding DFA state indeed contains that item (and only that item).

LR(0) Parsing (6)

Given a DFA recognising viable prefixes, an LR(0) parser can be constructed as follows:

- In a state without complete items: **Shift**
  - Read next terminal symbol and push it onto an internal parse stack.
  - Move to new state by following the edge labelled by the read terminal.
LR(0) Parsing (7)

- In a state with a **single complete item**: **Reduce**
  - The top of the parse stack contains the handle of the current right-sentential form (since we have recognised a viable prefix for which a single complete item is valid).
  - The handle is just the RHS of the valid item.
  - Reduce to the previous right-sentential form by replacing the handle on the parse stack with the LHS of the valid item.
  - Move to the state indicated by the new viable prefix on the parse stack.

LR(0) Parsing (8)

- If a state contains both complete and incomplete items, or if a state contains more than one complete item, then the grammar is not LR(0).

LR(0) Parsing (9)

```
<table>
<thead>
<tr>
<th>State</th>
<th>Stack (γ)</th>
<th>Input (w)</th>
<th>Move</th>
</tr>
</thead>
<tbody>
<tr>
<td>I0</td>
<td>e</td>
<td>abccde</td>
<td>Shift</td>
</tr>
<tr>
<td>I1</td>
<td>a</td>
<td>bcde</td>
<td>Shift</td>
</tr>
</tbody>
</table>
```

Note: $\gamma w$ is the current right-sentential form.

LR(0) Parsing (10)

```
<table>
<thead>
<tr>
<th>State</th>
<th>Stack (γ)</th>
<th>Input (w)</th>
<th>Move</th>
</tr>
</thead>
<tbody>
<tr>
<td>I2</td>
<td>ab</td>
<td>ccde</td>
<td>Shift</td>
</tr>
<tr>
<td>I3</td>
<td>abc</td>
<td>cde</td>
<td>Shift</td>
</tr>
<tr>
<td>I4</td>
<td>abcc</td>
<td>de</td>
<td>Reduce by $A \rightarrow c$</td>
</tr>
</tbody>
</table>
```
LR(0) Parsing (11)

State | Stack (γ) | Input (w) | Move
--- | --- | --- | ---
$I_7$ | abcA | de | Reduce by $A \rightarrow bcA$
$I_5$ | aA | de | Shift
$I_6$ | aAd | e | Reduce by $B \rightarrow d$

LR(0) Parsing (12)

State | Stack (γ) | Input (w) | Move
--- | --- | --- | ---
$I_8$ | aAB | e | Shift
$I_9$ | aABe | ε | Reduce by $S \rightarrow aABe$
$S$ | ε | Done

LR(0) Parsing (13)

Complete sequence ($γw$ is right-sentential form):

<table>
<thead>
<tr>
<th>State</th>
<th>Stack (γ)</th>
<th>Input (w)</th>
<th>Move</th>
</tr>
</thead>
<tbody>
<tr>
<td>$I_0$</td>
<td>$ε$</td>
<td>abccde</td>
<td>Shift</td>
</tr>
<tr>
<td>$I_1$</td>
<td>$a$</td>
<td>bccde</td>
<td>Shift</td>
</tr>
<tr>
<td>$I_2$</td>
<td>$ab$</td>
<td>ccde</td>
<td>Shift</td>
</tr>
<tr>
<td>$I_4$</td>
<td>$abc$</td>
<td>cde</td>
<td>Shift</td>
</tr>
</tbody>
</table>
| $I_3$ | $abcc$ | de | Reduce by $A \rightarrow c$
| $I_7$ | $abcA$ | de | Reduce by $A \rightarrow bcA$
| $I_5$ | $aA$ | de | Shift |
| $I_6$ | $aAd$ | e | Reduce by $B \rightarrow d$
| $I_8$ | $aAB$ | e | Shift |
| $I_9$ | $aABe$ | e | Reduce by $S \rightarrow aABe$
| $S$ | $ε$ | Done |

Cf: $S \Rightarrow aABe \Rightarrow aAde \Rightarrow abcAde \Rightarrow abccde$

LR(0) Parsing (14)

Even more clear that the parser carries out the rightmost derivation in reverse if we look at the right-sentential forms $γw$ of the reduction steps only:

\[
\begin{align*}
abccde & \leftarrow_{rm} \\
abcAde & \leftarrow_{rm} \\
aAde & \leftarrow_{rm} \\
aABe & \leftarrow_{rm} \\
S & \leftarrow_{rm}
\end{align*}
\]
LR Parsing & Left/Right Recursion (1)

Remark: Note how the right-recursive production

\[ A \rightarrow bcA \]

causes symbols \( bc \) to pile up on the parse stack until a reduction by

\[ A \rightarrow c \]

can occur, in turn allowing the stacked symbols to be reduced away.

LR(1) Grammars (1)

- In practice, LR(0) tends to be a bit too restrictive.
- If we add one symbol of “lookahead” by determining the set of terminals that possibly could follow a handle being reduced by a production \( A \rightarrow \beta \), then a wider class of grammars can be handled.
- Such grammars are called LR(1).

LR Parsing & Left/Right Recursion (2)

Even clearer if considering parsing of a string like

\[ abcbcbccde \text{ or } abcbcbcbcbccde \]

**Exercise:** Try parsing these!

Left-recursion allows reduction to happen sooner, thus keeping the size of the parse stack down. This is why left-recursive grammars often are preferred for LR parsing.

LR(1) Grammars (2)

Idea:

- Associate a lookahead set with items:

\[ A \rightarrow \alpha \cdot \beta, \{a_1, a_2, \ldots, a_n\} \]

- On reduction, a complete item is only valid if the next input symbol belongs to its lookahead set.
- Thus it is OK to have two or more simultaneously valid complete items, as long as their lookahead sets are disjoint.

(Similar to predictive recursive-descent parsing.)
LR($k$) Grammars

It is possible to have more than one symbol of lookahead:

- In general, a grammar that may be parsed with $k$ symbols of lookahead is called LR($k$).
- However, $k > 1$ does not add to the class of languages that can be defined.

LALR Grammars

A problem with LR($k$) parsers is that the DFAs become very large.

- LALR ("lookahead LR") is a simplified construction that leads to much smaller DFAs.
- The basic idea is to reduce the number of states by merging sets of LR(1) items that are "similar".
- LALR places some additional constraints on a grammar, but those constraints are not too severe in practice. Most programming languages have LALR grammars.