Bottom Up Parsing
Bottom-up Parsing: Shift-reduce parsing

Grammar H: \[ L \rightarrow E; L \mid E \]
\[ E \rightarrow a \mid b \]

Input: \(a;a;b\)

has parse tree

```
  L
 / \  /
E   ; L
 \   /  
a   a  b
```
Data for Shift-reduce Parser

- Input string: sequence of tokens being checked for grammatical correctness
- Stack: sentential form representing the input seen so far
- Trees Constructed: the parse trees that have been constructed so far at some point in the parsing
Operations of shift-reduce parser

• **Shift**: move input token to the set of trees as a singleton tree.
• **Reduce**: coalesce one or more trees into single tree (according to some production).
• **Accept**: terminate and accept the token stream as grammatically correct.
• **Reject**: Terminate and reject the token stream.
### A parse for grammar $H$

<table>
<thead>
<tr>
<th>Input</th>
<th>Stack</th>
<th>Action</th>
<th>Trees</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>a;a;b</code></td>
<td></td>
<td>Shift</td>
<td></td>
</tr>
<tr>
<td><code>;a;b</code></td>
<td><code>a</code></td>
<td>Reduce $E \rightarrow a$</td>
<td><code>a</code></td>
</tr>
<tr>
<td><code>;a;b</code></td>
<td><code>E</code></td>
<td>Shift</td>
<td><code>E</code></td>
</tr>
<tr>
<td><code>a;b</code></td>
<td><code>E;</code></td>
<td>Shift</td>
<td><code>E ;</code></td>
</tr>
</tbody>
</table>

Bottom-Up Parsing
### A parse for grammar H

<table>
<thead>
<tr>
<th>Input</th>
<th>Stack</th>
<th>Action</th>
<th>Trees</th>
</tr>
</thead>
<tbody>
<tr>
<td>; b</td>
<td>E; a</td>
<td>Reduce E → a</td>
<td>E ; a</td>
</tr>
<tr>
<td>; b</td>
<td>E ; E</td>
<td>Shift</td>
<td>E ; E</td>
</tr>
<tr>
<td>b</td>
<td>E ; E ;</td>
<td>Shift</td>
<td>E ; E ;</td>
</tr>
</tbody>
</table>

**Bottom-Up Parsing**
### A parse for grammar H

<table>
<thead>
<tr>
<th>Input</th>
<th>Stack</th>
<th>Action</th>
<th>Trees</th>
</tr>
</thead>
<tbody>
<tr>
<td>$</td>
<td>E;E;\text{;}b</td>
<td>Reduce E→\text{;}b</td>
<td>E ; E ; \text{;} b</td>
</tr>
<tr>
<td></td>
<td>E;E;E</td>
<td>Reduce \text{;}L→E</td>
<td>E ; E ; \text{;} E</td>
</tr>
<tr>
<td></td>
<td>E;E;L</td>
<td>Reduce \text{;}L→E;L</td>
<td>E ; E ; \text{;} L</td>
</tr>
</tbody>
</table>

Bottom-Up Parsing
A parse for grammar $H$

<table>
<thead>
<tr>
<th>Input</th>
<th>Stack</th>
<th>Action</th>
<th>Trees</th>
</tr>
</thead>
<tbody>
<tr>
<td>$</td>
<td>$</td>
<td>Reduce $L \to E;L$</td>
<td>$</td>
</tr>
<tr>
<td></td>
<td>$L$</td>
<td></td>
<td>$L$</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$</td>
<td>$L$</td>
<td>Acc</td>
<td>$</td>
</tr>
</tbody>
</table>

$L$ $L$ $b$

$E$ $;$$E$ $;$$E$ $;$$E$ $;$$E$

$E$ $;$$E$ $;$$E$ $;$$E$ $;$$E$

$E$ $;$$E$ $;$$E$ $;$$E$ $;$$E$

$E$ $;$$E$ $;$$E$ $;$$E$ $;$$E$

$E$ $;$$E$ $;$$E$ $;$$E$ $;$$E$

$E$ $;$$E$ $;$$E$ $;$$E$ $;$$E$

$E$ $;$$E$ $;$$E$ $;$$E$ $;$$E$

$E$ $;$$E$ $;$$E$ $;$$E$ $;$$E$
Bottom-up Parsing

• Characteristic automata for an “LR” grammar: tells when to shift/reduce/accept or reject
• handles
• viable prefixes
## A parse for grammar H

<table>
<thead>
<tr>
<th>Input</th>
<th>Stack</th>
<th>Action</th>
<th>Stack+Input</th>
</tr>
</thead>
<tbody>
<tr>
<td>a;a;b</td>
<td></td>
<td>Shift</td>
<td>a;a;b</td>
</tr>
<tr>
<td>;a;b</td>
<td>a</td>
<td>Reduce E→ a</td>
<td>a;a;b</td>
</tr>
<tr>
<td>;a;b</td>
<td>E</td>
<td>Shift</td>
<td>E;a;b</td>
</tr>
<tr>
<td>a;b</td>
<td>E;</td>
<td>Shift</td>
<td>E;a;b</td>
</tr>
<tr>
<td>;b</td>
<td>E; a</td>
<td>Reduce E→ a</td>
<td>E;a;b</td>
</tr>
<tr>
<td>;b</td>
<td>E;E</td>
<td>Shift</td>
<td>E;E;b</td>
</tr>
<tr>
<td>b</td>
<td>E;E;</td>
<td>Shift</td>
<td>E;E;b</td>
</tr>
<tr>
<td>$</td>
<td>E;E;b</td>
<td>Reduce E→ b</td>
<td>E;E;b</td>
</tr>
<tr>
<td>$</td>
<td>E;E;E</td>
<td>Reduce L→ E</td>
<td>E;E;E</td>
</tr>
<tr>
<td>$</td>
<td>E;E;L</td>
<td>Reduce L→ E;L</td>
<td>E;E;L</td>
</tr>
<tr>
<td>$</td>
<td>E;L</td>
<td>Reduce L→ E;L</td>
<td>E;L</td>
</tr>
<tr>
<td>$</td>
<td>L</td>
<td>Accept</td>
<td>L</td>
</tr>
</tbody>
</table>
“Bottom-up” parsing?

Stack+Input

L
E;L
E;E;L
E;E;E
E;E;b
E;E;b
E;a;b
E;a;b
E;a;b
Handles and viable prefixes

- Stack + remaining input = sentential form
- Handle = the part of the sentential form that is reduced in each step
- Viable prefix = the prefix of the sentential form in a right-most derivation that do not extend beyond the end of the handle
- E.g. viable prefixes for H: \((E;)*(E \mid L \mid a \mid b)\)
- Viable prefixes form a regular set.
Characteristic Finite State Machine (CFSM)

Viable prefixes of H are recognized by this CFSM:

```
Viable prefixes of H are recognized by this CFSM:
```

![Finite State Machine Diagram]

```
Characteristic Finite State Machine (CFSM)
```

```
Viable prefixes of H are recognized by this CFSM:
```

```
Characteristic Finite State Machine (CFSM)
```

```
Viable prefixes of H are recognized by this CFSM:
```

```
Characteristic Finite State Machine (CFSM)
```

```
Viable prefixes of H are recognized by this CFSM:
```

```
Characteristic Finite State Machine (CFSM)
```

```
Viable prefixes of H are recognized by this CFSM:
```

```
Characteristic Finite State Machine (CFSM)
```

```
Viable prefixes of H are recognized by this CFSM:
```

```
Characteristic Finite State Machine (CFSM)
```

```
Viable prefixes of H are recognized by this CFSM:
```

```
Characteristic Finite State Machine (CFSM)
```

```
Viable prefixes of H are recognized by this CFSM:
```

```
Characteristic Finite State Machine (CFSM)
```

```
Viable prefixes of H are recognized by this CFSM:
```

```
Characteristic Finite State Machine (CFSM)
```

```
Viable prefixes of H are recognized by this CFSM:
```

```
Characteristic Finite State Machine (CFSM)
```

```
Viable prefixes of H are recognized by this CFSM:
```

```
Characteristic Finite State Machine (CFSM)
```

```
Viable prefixes of H are recognized by this CFSM:
```

```
Characteristic Finite State Machine (CFSM)
```

```
Viable prefixes of H are recognized by this CFSM:
```

```
Characteristic Finite State Machine (CFSM)
```

```
Viable prefixes of H are recognized by this CFSM:
```

```
Characteristic Finite State Machine (CFSM)
```

```
Viable prefixes of H are recognized by this CFSM:
```

```
Characteristic Finite State Machine (CFSM)
```

```
Viable prefixes of H are recognized by this CFSM:
```

```
Characteristic Finite State Machine (CFSM)
```

```
Viable prefixes of H are recognized by this CFSM:
```

```
Characteristic Finite State Machine (CFSM)
```
How a Bottom-Up Parser Works

• Run the CFSM on symbols in the stack
• If a transition possible on the incoming input symbol, then shift, else reduce.
  – Still need to decide which rule to use for the reduction.
Characteristic automaton

(start) 0 → 1
E 2 → 5
L a 3 → 4
b

E 2 → 5
E 5 → 6
L

E;E;E leads to state 2 after E;E;E
E;E;L leads to state 6 after E;E;L
E;L leads to state 6 after E;L

Viable Prefixes

a;a;b leads to state 3 after a
E;a;b leads to state 3 after E;a
E;E;b leads to state 4 after E;E;b
E;E;E leads to state 2 after E;E;E
E;E;L leads to state 6 after E;E;L
E;L leads to state 6 after E;L
Characteristic automaton

**State** | **Action** |
--- | --- |
0,5 | shift (if possible) |
1 | accept |
2 | reduce L→E, if EOF, shift otherwise |
3 | reduce E→a |
4 | reduce E→b |
6 | reduce L→E;L |

Bottom-Up Parsing
Example: expression grammar

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

\[id + id + id + id\]

has parse tree:
A parse in this grammar

\[
\begin{align*}
\text{id+id+id+id} & \quad \text{Shift} \\
\text{id+id+id} & \quad \text{id} \quad \text{Reduce} \\
\text{id+id+id} & \quad \text{P} \quad \text{Reduce} \\
\text{id+id+id} & \quad \text{T} \quad \text{Reduce} \\
\text{id+id+id} & \quad \text{E} \quad \text{Reduce} \\
\text{id+id+id} & \quad \text{E} \quad \text{Shift} \\
\text{id+id+id} & \quad \text{E+} \quad \text{Reduce} \\
\text{id+id+id} & \quad \text{E+P} \quad \text{Reduce} \\
\text{id+id+id} & \quad \text{E+T} \quad \text{Reduce} \\
\text{id+id+id} & \quad \text{E} \quad \text{Shift} \\
\text{id+id+id} & \quad \text{E+} \quad \text{Shift}
\end{align*}
\]
A parse in this grammar (cont.)

+id       E+ id       Reduce
+id       E+P        Reduce
+id       E+T        Reduce
+id       E          Shift
id        E+         Shift
$          E+id      Reduce
$          E+P       Reduce
$          E+T       Reduce
$          E         Reduce
$          E         Accept
The CFSM recognizes viable prefixes (strings of grammar symbols that can appear on the stack):
Definitions

- Rightmost derivation
- Right-sentential form
- Handle
- Viable prefix
- Characteristic automaton
**Rightmost Derivation**

**Definition:** A *rightmost derivation* in $G$ is a derivation:

$$S \Rightarrow w_1 \Rightarrow \ldots \Rightarrow w_i \Rightarrow w_{i+1} \Rightarrow \ldots$$

such that for each step $i$, the rightmost non-terminal in $w_i$ is replaced to obtain $w_{i+1}$

1. $L \Rightarrow L; E \Rightarrow L; E; E \Rightarrow E; E; E \Rightarrow a; E; E \Rightarrow a; a; E \Rightarrow a; a; b$
   is not right-most

2. $L \Rightarrow L; E \Rightarrow L; b \Rightarrow L; E; b \Rightarrow L; a; b \Rightarrow E; a; b \Rightarrow a; a; b$
   is right-most
Right-sentential Forms

Definition: A right-sentential form is any sentential form that occurs in a right-most derivation.

L ⇒ L; E ⇒ L; b ⇒ L; E; b ⇒ L; a; b ⇒ E; a; b ⇒ a; a; b

E.g., any of these
Definition: Assume the $i$-th step of a rightmost derivation is:

$$w_i = u_i Av_i \Rightarrow u_i \alpha v_i = w_{i+1}$$

Then, $(\alpha, |u_i\alpha|)$ is the handle of $w_{i+1}$

In an unambiguous grammar, any sentence has a unique rightmost derivation, and so we can talk about “the” handle rather than “a” handle.
The Plan

• Construct a parser by first constructing the CFSM, then constructing GOTO and ACTION tables from it.

• Construction has two parts:
  – “LR(0)” construction of the CFSM
  – “SLR(1)” construction of tables
The states in the CFSM are created by taking the “closure” of “LR(0) items”

Given a production “L → E ; L”, these are all induced “LR(0) items”

\[
L \rightarrow \cdot E ; L \\
L \rightarrow E \cdot ; L \\
L \rightarrow E ; \cdot L \\
L \rightarrow E ; L \cdot
\]
What is “•”?

The “•” in “L → E • ; L” represents the state of the parse.

Only this part of the tree is fully developed.
A State in the CFSM: Closure of LR(0) Item

For set I of LR(0) items, calculate closure(I):
1. if “A → α • B β” is in closure(I), then for every production “B → γ”, “B → • γ” is in closure(I)
2. closure(I) is the smallest set with property (1)
Closure of LR(0) Item: Example

H: \( L \rightarrow E; L \mid E \)

\[
E \rightarrow a \mid b
\]

closure(\{L \rightarrow E ; \cdot L\}) =
\{L \rightarrow E ; \cdot L,
L \rightarrow \cdot E ; L ,
L \rightarrow \cdot E ,
E \rightarrow \cdot a ,
E \rightarrow \cdot b\}
LR(0) Machine

- Given grammar \( G \) with goal symbol \( S \), augment the grammar by adding new goal symbol \( S' \) and production \( S' \rightarrow S \).
- States = sets of LR(0) items
- Start state = \( \text{closure}({S' \rightarrow \cdot S}) \)
- All states are considered to be final (set of viable prefixes closed under prefix)
- transition\((I, X) = \text{closure}({A \rightarrow \alpha X \cdot \beta \mid A \rightarrow \alpha \cdot X \beta \in I})\).
**Example: LR(0) CFSM Construction**

Augment the grammar:

\[
\begin{align*}
H: & \quad L' \rightarrow L \\
    & \quad L \rightarrow E; L \mid E \\
    & \quad E \rightarrow a \mid b
\end{align*}
\]

Initial State is closure of this “augmenting rule”:

\[
I_0: \begin{array}{|l|}
\hline
L' \rightarrow \cdot L \\
L \rightarrow \cdot E; L \\
L \rightarrow \cdot E \\
E \rightarrow \cdot a \\
E \rightarrow \cdot b \\
\hline
\end{array}
\]

\[
closure(\{L' \rightarrow \cdot L\}) =
\]
Example: Transitions from I₀

\[
\text{transition}(I₀, L) = \{L' \rightarrow L \cdot\} = I₁
\]

\[
\text{transition}(I₀, E) = \{L \rightarrow E \cdot; L, L \rightarrow E \cdot\} = I₂
\]

\[
\text{transition}(I₀, a) = \{E \rightarrow a \cdot\} = I₃
\]

\[
\text{transition}(I₀, b) = \{E \rightarrow b \cdot\} = I₄
\]

There are no other transitions from I₀.
There are no transitions possible from I₁.
Now consider the transitions from I₂.
Transitions from \( I_2 \)

\[ I_5 : \begin{align*}
    &L \rightarrow E; \cdot L \\
    &L \rightarrow \cdot E; L \\
    &L \rightarrow \cdot E \\
    &E \rightarrow \cdot a \\
    &E \rightarrow \cdot b
\end{align*} \]

transition( \( I_2, ; \) ) =

New state: 

\[ \text{transition}( I_5, L) = \{ L \rightarrow E; L\cdot \} = I_6 \]

\[ \text{transition}( I_5, E) = I_2 \]

\[ \text{transition}( I_5, a) = I_3 \quad \text{transition}( I_5, b) = I_4 \]
The CFSM Transition Diagram for H
Characteristic Finite State Machine for H

Bottom-Up Parsing
How LR(1) parsers work

• GOTO table: transition function of characteristic automaton in tabular form
• ACTION table: State × Σ → Action
• Procedure:
  – Use the GOTO table to run the CFSM over the stack. Suppose state reached at top of stack is σ.
  – Take action given by ACTION(σ,a), where a is incoming input symbol.
### Action Table for H

<table>
<thead>
<tr>
<th></th>
<th>a</th>
<th>b</th>
<th>;</th>
<th>$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Shift</td>
<td>Shift</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td>Accept</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>Shift</td>
<td>Reduce L→E</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>Reduce E→a</td>
<td>Reduce E→a</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>Reduce E→b</td>
<td>Reduce E→b</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>Shift</td>
<td>Shift</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
<td>Reduce L→E;L</td>
<td></td>
</tr>
</tbody>
</table>
SLR(1) Parser Construction

• GOTO table is the move function from the LR(0) CFSM.

• ACTION table is constructed from the CFSM as follows:
  – If state i contains $A \rightarrow \alpha \cdot a \beta$, then ACTION(i,a) = Shift.
  – If state i contains $A \rightarrow \alpha \cdot$, then ACTION(i,a) = Reduce $A \rightarrow \alpha$ (But, if $A$ is $L'$, action is Accept.)
  – Otherwise, reject.

• But,...
SLR(1) Parser Construction

- Rules for the ACTION table can involve *shift/reduce conflicts*.
- So the actual rule for reduce actions is:
  
  If state i contains $A \rightarrow \alpha \cdot$, then $\text{ACTION}(i,a) = \text{Reduce } A \rightarrow \alpha$, for all $a \in \text{FOLLOW}(A)$.

- E.g. state 2 for grammar H yields a shift/reduce conflict. Namely, should you shift the “;” or reduce by “$L \rightarrow E$”. This is resolved by looking at the “follow set” for L.

- $\text{Follow}(L) = \{$$\}$
**FIRST and FOLLOW sets**

- \( \text{FIRST}(\alpha) = \{a \in \Sigma \mid \alpha \Rightarrow^* a\beta\} \cup \{\Lambda \mid a \Rightarrow^* \Lambda\} \)
- \( \text{FOLLOW}(A) = \{a \in \Sigma \mid S \Rightarrow^* \alpha A \alpha \beta\} \)
Calculating FIRST sets

- Create table $Fi$ mapping $N$ to $\sum \cup \{\Lambda\}$; initially, $Fi(A) = \emptyset$ for all $A$.
- Repeat until $Fi$ does not change:
  - For each $A \rightarrow \alpha \in P$,
    $$Fi(A) := Fi(A) \cup FIRST(\alpha, Fi)$$

where $FIRST(\alpha, Fi)$ is defined as follows:
- $FIRST(\Lambda, Fi) = \{\Lambda\}$
- $FIRST(a\beta, Fi) = \{a\}$
- $FIRST(B\beta, Fi) = Fi(B) \cup FIRST(b, Fi)$, if $\Lambda \in Fi(B)$
  $$Fi(B), \text{ o.w.}$$
Calculating FOLLOW sets

• Calculate FOLLOW sets
• Create table $F_o : N \rightarrow \sum \cup \{\$\};$ initially, $F_o(A) = \emptyset$ for all $A,$ except $F_o(S) = \{\$\}$
• Repeat until $F_o$ does not change:
  – For each production $A \rightarrow \alpha B\beta,$
    $F_o(B) := F_o(B) \cup FIRST(\beta) - \{\lambda\}$
  – For each production $A \rightarrow \alpha B,$
    $F_o(B) := F_o(B) \cup F_o(A)$
  – For each production $A \rightarrow \alpha B\beta,$ if $\lambda \in FIRST(\beta),$
    $F_o(B) := F_o(B) \cup F_o(A)$